

# API 开发指南

——Autodesk® Revit®

宦国胜 主编



中国水利水电出版社  
www.waterpub.com.cn

# API 开发指南

——Autodesk® Revit®

宦国胜 主编



中国水利水电出版社  
[www.waterpub.com.cn](http://www.waterpub.com.cn)

· 北京 ·



## 内 容 提 要

本书由江苏省水利勘测设计研究院有限公司数字工程部工程师们在 Revit API 开发实战过程中参照 Revit API 在线英文帮助文献翻译编写,系统介绍开发人员如何应用 Revit API 接口进行 Revit 的二次开发。

本书以 Revit 2014 版软件为平台,介绍 Revit API 的基础知识、开发工程和相关资源。书中配以大量的实例代码、图片和表格,方便读者更好地理解相关内容。参照本书 Revit API 知识,读者可以搭建二次开发环境,开发 Revit 插件实现 Revit 数据的读取、创建、修改、导入导出等;也可以通过 API 和 .NET 技术来创建用户交互界面,提供更好的用户体验;通过扩展 Revit 本身尚不具备的功能,使得 Revit 和其他软件平台进行交互,实现数据验证、检查和操作自动化,极大地提高数据利用率和设计效率。

本书适用于 Revit API 开发人员和相关高等院校师生,既可作为 Revit API 初学者的入门指南,也可供具备一定 API 编程经验的开发人员参考。

## 图书在版编目(CIP)数据

API开发指南: Autodesk®Revit® / 宦国胜主编. --  
北京: 中国水利水电出版社, 2016. 12  
ISBN 978-7-5170-5000-1

I. ①A… II. ①宦… III. ①建筑设计—计算机辅助  
设计—应用软件—指南 IV. ①TU201.4-62

中国版本图书馆CIP数据核字(2016)第316913号

书 名	<b>API 开发指南</b> ——Autodesk® Revit® API KAIFA ZHINAN
作 者	宦国胜 主编
出版发行	中国水利水电出版社 (北京市海淀区玉渊潭南路1号D座 100038) 网址: www.waterpub.com.cn E-mail: sales@waterpub.com.cn 电话: (010) 68367658 (营销中心)
经 售	北京科水图书销售中心(零售) 电话: (010) 88383994、63202643、68545874 全国各地新华书店和相关出版物销售网点
排 版	中国水利水电出版社微机排版中心
印 刷	北京嘉恒彩色印刷有限责任公司
规 格	184mm×260mm 16开本 29.75印张 705千字
版 次	2016年12月第1版 2016年12月第1次印刷
印 数	0001—2000册
定 价	<b>90.00</b> 元

凡购买我社图书,如有缺页、倒页、脱页的,本社营销中心负责调换

版权所有·侵权必究

## 本书编委会

主 编 宦国胜

编 委 沈国华 贾 健 张 耘 王海俊

左威龙 张 超 吝江峰 徐 鹏

陈蕾蕾 洪项华

# 序

建筑信息模型（Building Information Modeling, BIM）是以三维数字技术为基础，集成建筑工程项目各种相关信息的工程数据模型，通过数字信息仿真模拟建筑物所具有的真实信息。

BIM 理念提出至今已经 40 余年，经历了萌芽、产生和发展 3 个阶段。由于 BIM 具有可视化、协调性、模拟性、优化性和可出图性五大特点，目前 BIM 概念在建筑领域已深入人心，发展异常迅猛，在全球范围内得以推广应用，在我国越来越多的基础设施类工程也开始逐渐应用。尤其是当 BIM 被明确写入我国建筑业发展“十二五”规划，并继续列入住房和城乡建设部、科学技术部“十三五”相关规划之后，BIM 发展趋势更是势不可挡。BIM 技术可以说是我国建筑类从设计到运行维护全生命周期的一次技术进步和革命。

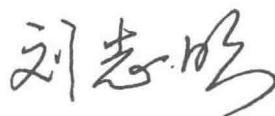
借助 BIM 技术，未来的设计将不再是单纯意义上的设计交流、组织及管理手段，它将与 BIM 融合，贯穿设计全过程，成为设计手段本身的一部分。借助于 BIM 的技术优势，协同的范畴也将从单纯的设计阶段扩展到建筑全生命周期，需要设计、施工、运营、维护等各方的集体参与，因此具备了更广泛的意义，从而带来综合效率的大幅提升。

Autodesk Revit 是一款基于先进的三维数字设计解决方案应用于工程设计建造管理的数据化工具，但主要面向房建类工程，对基础设施行业（包括水利、交通等）尚缺乏专业上的支持，给 BIM 的全面推广应用带来了诸多不便，这就需要依赖各行各业结合各自的特点开展针对性的专项研究，除了应用层面的研究，还需要开展大量基础性的二次开发，这样才能真正意义上推动国内建筑类的 BIM 发展。

目前 BIM 技术应用类的基础教程较多，但二次开发方面的书籍还非常缺乏。江苏省水利勘测设计研究院有限公司的 BIM 团队，基于多年来在 Revit 平台应用和二次开发的基础上，整理汇编《API 开发指南——Autodesk® Revit®》一书，对从事建筑、结构、MEP 各专业相关的 API 开发技术人员、土建类院校的教授和学生有着重要的借鉴作用。中国水利水电 BIM 设计联盟于 2016 年 10 月正式

成立，该书的出版恰逢其时，为广大协会成员提供了一本实用性强的技术指南。希望该书有助于培养一批专业的 BIM 二次开发人才，为我国 BIM 技术的普及和提高起到积极的推动作用。

水利部水利水电规划设计总院副院长  
中国水利水电勘测设计协会理事长



2016 年 12 月

# 前言

Autodesk Revit 作为目前国内外应用广泛的 BIM 软件,除了自身功能强大之外,同时提供丰富的应用程序编程接口 API (Application Programming Interface)。用户可以通过 API 来集成其他应用程序或者使用 API 来二次开发自己的应用程序操纵和访问 Revit,把琐碎的建模工作自动化,自动检查 Revit 文件中的错误,借助 API 把其他软件的功能集成或链接到 Revit 中来,执行各种分析,在一个平台上完成用户的需求。

本书目录和标题采用中英文对照的方式,主要是为了保证叙述的准确性和唯一性。代码部分沿用 IT 行业习惯全部采用英文,对于一些组合词,无法翻译成中文的英文(如类名、函数名等)也沿用英文。另外,部分代码段的标题,为了便于阅读和对应查找也保留了英文。

本书所述主要以 Revit 2014 为主体,少量提及之前的版本,2014 之后版本的相关技术请参考相应的更新文献。本书技术内容操作性要求比较高,由于时间和精力所限,只对其中我们感兴趣和工作过程中涉及的代码进行了上机调试和运行。

本书适用于熟悉 Revit 应用的 API 开发技术人员和土木类院校师生,既可作为 Revit API 初学者的入门指南,也可作为具备一定 API 编程经验的开发人员的参考手册。

本书涵盖 Revit 建筑、结构、水暖电等方面的 API 开发内容,共分为 6 章。

第 1 章引言部分为全书的总览,介绍了 Revit API 基本功能和必要的基础知识,包含一些基础演练及代码,应用程序和文件,Revit 图元基本分类、检索及属性等。

第 2 章 Revit 图元基本交互部分为全书的基础,介绍了图元的过滤、选集、参数、集合、图元的编辑以及各种视图等。

第 3 章 Revit 几何图元部分为图元进阶专题,介绍了房屋建筑相关的几何图元,族实例和族文件,概念设计,基准和信息图元,注释图元,几何类,二维、三维草图类及模型曲线,材料信息及管理。

第 4 章规程特有功能部分为产品特有 API,分别介绍了 Architecture 中的房间相关功能;Structure 中的结构模型、分析模型、荷载,分析应用程序链接,

分析节点图元连接；MEP 中的暖通空调和管道，数据访问，包括专有图元、族创建，机械、电气设置及布管系统配置等。

第 5 章进阶专题部分为二次开发所必需的重点章节，包括共享参数，事务、事件，外部事件及可停靠对话框，动态模型更新，点云、分析、工作共享、导出等。

附录部分包括常用术语解释，常见问题解答，如何用 Visual Basic .NET 创建应用程序以及 API 用户界面指南。

本书编委会成员多为从事 Revit API 开发和测试的工程师。全书由宦国胜主笔并最终统稿，沈国华、贾健负责全书的校对等工作。其中，沈国华、贾健参与了第 1 章编写，吝江峰参与了第 2 章编写，左威龙参与了第 3 章编写，张超参与了第 4 章编写，王海俊参与了第 5 章编写，张耘、徐鹏、陈蕾蕾、洪项华参与了附录部分编写。

本书参考了 Revit 帮助中的主要内容，示例代码大部分来源于 Revit 软件开发工具包（SDK）。有兴趣的读者，可通过访问相关网站学习更多 Revit API 相关的知识。

本书中代码的调试需在 Revit 2014 版的软件环境下，且系统需安装 Microsoft.NET Framework 4.0（或更新版本）。编程语言可选用 .NET 框架协议下的 C# 或 Visual Basic.NET 等某一兼容编程语言。

在本书的编写整理过程中，虽经反复斟酌，但由于编者水平所限，加之时间紧迫，错误和疏漏在所难免，敬请读者批评指正。

最后，由衷感谢 Autodesk 公司产品销售经理李忠、Autodesk 公司软件研发有限公司卢石碧的鼎力支持和帮助。感谢编委会成员的不懈努力，在百忙之中抽出时间做研究、测试并编写修改本书。

作者

2016 年 12 月



# 目 录

## 序 前言

第 1 章 引言 (Introduction)	1
1.1 欢迎使用 Revit 平台 API (Welcome to the Revit Platform API)	1
1.1.1 Revit 平台 API 简介 (Introduction to the Revit Platform API)	1
1.1.2 Revit 平台 API 能做什么 (What Can You Do with the Revit Platform API)	1
1.1.3 必要条件 (Requirements)	2
1.1.4 安装 (Installation)	2
1.1.5 受支持的编程语言 (Supported Programming Languages)	2
1.1.6 使用手册 (User Manual)	2
1.1.7 文档协定 (Documentation Conventions)	4
1.1.8 本版本的新特性	4
1.2 入门 (Getting Started)	4
1.2.1 演练 (Walkthroughs)	4
1.2.2 演练: Hello World (Walkthrough: Hello World)	4
1.2.3 演练: 添加 Hello World 功能区面板 (Add Hello World Ribbon Panel)	10
1.2.4 演练: 检索所选图元 (Retrieve Selected Elements)	13
1.2.5 演练: 检索过滤图元 (Retrieve Filtered Elements)	14
1.3 插件集成 (Add-in Integration)	15
1.3.1 概述 (Overview)	15
1.3.2 外部命令 (External Commands)	15
1.3.3 外部应用程序 (External Application)	20
1.3.4 注册插件 (Add-in Registration)	21
1.3.5 本地化 (Localization)	26
1.3.6 属性 (Attributes)	26
1.3.7 Revit 异常 (Revit Exceptions)	28
1.3.8 功能区面板和控件 (Ribbon Panels and Controls)	28
1.3.9 Revit 式任务对话框 (Revit-style Task Dialogs)	37
1.3.10 数据库级外部应用程序 (DB-level External Applications)	39
1.4 应用程序和文件 (Application and Document)	40
1.4.1 应用程序功能 (Application Functions)	40

1.4.2	文件功能 (Document Functions)	43
1.4.3	文档及文件管理 (Document and File Management)	44
1.4.4	设置 (Settings)	47
1.4.5	单位 (Units)	48
1.5	图元概要 (Elements Essentials)	51
1.5.1	图元分类 (Element Classification)	51
1.5.2	其他分类 (Other Classifications)	52
1.5.3	图元检索 (Element Retrieval)	56
1.5.4	通用属性 (General Properties)	57
<b>第 2 章</b>	<b>Revit 图元基本交互 (Basic Interaction with Revit Elements)</b>	<b>61</b>
2.1	过滤 (Filtering)	61
2.1.1	创建图元过滤集 (Create a FilteredElementCollector)	61
2.1.2	应用过滤器 (Applying Filters)	62
2.1.3	获取过滤图元或图元 ID (Getting Filtered Elements or Element IDs)	68
2.1.4	LINQ 查询 (LINQ Queries)	71
2.1.5	边界框过滤器 (Bounding Box Filters)	72
2.1.6	图元相交过滤器 (Element Intersection Filters)	72
2.2	选集 (Selection)	73
2.2.1	更改选集 (Changing the Selection)	74
2.2.2	用户选集 (User Selection)	75
2.2.3	过滤的用户选集 (Filtered User Selection)	77
2.3	参数 (Parameters)	78
2.3.1	演练: 获取所选图元参数 (Walkthrough: Get Selected Element Parameters)	79
2.3.2	定义 (Definition)	81
2.3.3	内建参数 (BuiltInParameter)	82
2.3.4	存储类型 (StorageType)	83
2.3.5	AsValueString()和 SetValueString()	84
2.3.6	参数关系 (Parameter Relationships)	85
2.3.7	给图元添加参数 (Adding Parameters to Elements)	86
2.4	集合 (Collections)	86
2.4.1	接口 (Interface)	86
2.4.2	集合和迭代器 (Collections and Iterators)	87
2.5	编辑图元 (Editing Elements)	88
2.5.1	移动图元 (Moving Elements)	89
2.5.2	复制图元 (Copying Elements)	91
2.5.3	旋转图元 (Rotating Elements)	92
2.5.4	对齐图元 (Aligning Elements)	93
2.5.5	镜像图元 (Mirroring Elements)	94

2.5.6	成组图元 (Grouping Elements)	94
2.5.7	创建图元阵列 (Creating Arrays of Elements)	95
2.5.8	删除图元 (Deleting Elements)	96
2.5.9	锁定图元 (Pinned Elements)	97
2.6	视图 (Views)	98
2.6.1	关于视图 (About Views)	98
2.6.2	视图类型 (View Types)	101
2.6.3	视图过滤器 (View Filters)	125
2.6.4	视图裁剪 (View Cropping)	126
2.6.5	位移视图 (Displaced Views)	126
2.6.6	用户界面视图 (UIView)	127

### 第3章 Revit 几何图元 (Revit Geometric Elements) 129

3.1	墙、楼板、天花板、屋顶和洞口 (Walls, Floors, Ceilings, Roofs and Openings)	129
3.1.1	墙 (Walls)	129
3.1.2	楼板、天花板和基础 (Floors, Ceilings and Foundations)	131
3.1.3	屋顶 (Roofs)	133
3.1.4	幕墙 (Curtains)	135
3.1.5	其他图元 (Other Elements)	135
3.1.6	复合结构 (CompoundStructure)	135
3.1.7	洞口 (Opening)	138
3.1.8	热属性 (Thermal Properties)	140
3.2	族实例 (Family Instances)	142
3.2.1	识别图元 (Identifying Elements)	142
3.2.2	族 (Family)	143
3.2.3	族实例 (FamilyInstances)	143
3.2.4	代码示例 (Code Samples)	150
3.2.5	族符号 (FamilySymbol)	155
3.3	族文件 (Family Documents)	156
3.3.1	关于族文件 (About Family Documents)	156
3.3.2	在族中创建图元 (Creating Elements in Families)	157
3.3.3	族图元的可见性 (Visibility of Family Elements)	162
3.3.4	管理族类型和参数 (Managing Family Types and Parameters)	163
3.4	概念设计 (Conceptual Design)	165
3.4.1	点和曲线对象 (Point and Curve Objects)	165
3.4.2	形状 (Forms)	168
3.4.3	有理化处理表面 (Rationalizing a Surface)	173
3.4.4	自适应构件 (Adaptive Components)	178

3.5 基准和信息图元 (Datum and Information Elements)	178
3.5.1 标高 (Levels)	179
3.5.2 轴网 (Grids)	181
3.5.3 阶段 (Phase)	183
3.5.4 设计选项 (Design Options)	184
3.6 注释图元 (Annotation Elements)	185
3.6.1 尺寸和限制条件 (Dimensions and Constraints)	185
3.6.2 详图曲线 (Detail Curve)	190
3.6.3 标记 (Tags)	190
3.6.4 文本 (Text)	192
3.6.5 注释符号 (Annotation Symbol)	193
3.7 几何 (Geometry)	193
3.7.1 示例: 检索墙的几何数据 (Example: Retrieve Geometry Data from a Wall)	194
3.7.2 几何对象类 (GeometryObject Class)	195
3.7.3 几何助手类 (Geometry Helper Classes)	213
3.7.4 集合类 (Collection Classes)	220
3.7.5 示例: 检索梁的几何数据 (Example: Retrieve Geometry Data from a Beam)	221
3.7.6 体拉伸分析 (Extrusion Analysis of a Solid)	222
3.7.7 由光线投影找出几何体 (Finding Geometry by Ray Projection)	224
3.7.8 几何实用程序类 (Geometry Utility Classes)	228
3.7.9 房间和空间几何对象 (Room and Space Geometry)	229
3.8 草图 (Sketching)	231
3.8.1 二维草图 (The 2D Sketch Class)	232
3.8.2 三维草图 (3D Sketch)	234
3.8.3 模型曲线 (ModelCurve)	240
3.9 材料 (Material)	242
3.9.1 一般材料信息 (General Material Information)	242
3.9.2 材料管理 (Material Management)	244
3.9.3 图元材料 (Element Material)	246
3.9.4 材料数量 (Material Quantities)	252
3.9.5 涂装图元表面 (Painting the Face of an Element)	252
3.10 楼梯和栏杆扶手 (Stairs and Railings)	253
3.10.1 楼梯 (Stairs)	253
3.10.2 栏杆扶手 (Railings)	257
3.10.3 楼梯注释 (Stairs Annotations)	258
3.10.4 楼梯构件 (Stairs Components)	260
<b>第4章 规程特有功能 (Discipline-Specific Functionality)</b>	<b>265</b>
4.1 Revit Architecture	265

4.2	Revit Structure	275
4.2.1	结构模型图元 (Structural Model Elements)	275
4.2.2	分析模型 (Analytical Model)	284
4.2.3	荷载 (Loads)	292
4.2.4	分析链接 (Analysis Link)	294
4.2.5	分析连接 (Analytical Links)	295
4.3	Revit MEP	297
4.3.1	MEP 图元创建 (MEP Element Creation)	297
4.3.2	连接件 (Connectors)	303
4.3.3	族创建 (Family Creation)	305
4.3.4	机械设置 (Mechanical Settings)	306
4.3.5	电气设置 (Electrical Settings)	311
4.3.6	布管系统配置 (Routing Preferences)	313
<b>第 5 章</b>	<b>进阶专题 (Advanced Topics)</b>	<b>315</b>
5.1	在 Revit 模型中存储数据 (Storing Data in the Revit Model)	315
5.1.1	共享参数 (Shared Parameters)	315
5.1.2	定义文件 (Definition File)	315
5.1.3	绑定 (Binding)	319
5.1.4	可扩展存储 (Extensible Storage)	322
5.2	事务 (Transactions)	325
5.2.1	事务类 (Transaction Classes)	325
5.2.2	事件中的事务 (Transactions in Events)	329
5.2.3	故障处理选项 (Failure Handling Options)	330
5.2.4	获取图元几何和分析模型 (Getting Element Geometry and Analytical Model)	331
5.2.5	临时事务 (Temporary Transactions)	332
5.3	事件 (Events)	332
5.3.1	数据库事件 (Database Events)	333
5.3.2	用户界面事件 (User Interface Events)	334
5.3.3	注册事件 (Registering Events)	334
5.3.4	取消事件 (Canceling Events)	335
5.4	外部事件 (External Events)	336
5.5	可停靠对话框 (Dockable Dialog Panes)	339
5.6	动态模型更新 (Dynamic Model Update)	340
5.6.1	实现更新器接口 (Implementing Iupdater)	340
5.6.2	Execute 方法 (The Execute Method)	343
5.6.3	注册更新器 (Registering Updaters)	344
5.6.4	接触最终用户 (Exposure to End-User)	345

5.7 命令 (Commands) .....	347
5.8 故障发布和处理 (Failure Posting and Handling) .....	350
5.8.1 发布故障 (Posting Failures) .....	350
5.8.2 处理故障 (Handling Failures) .....	354
5.9 性能顾问 (Performance Adviser) .....	359
5.10 点云 (Point Clouds) .....	363
5.10.1 点云客户端 (Point Cloud Client) .....	363
5.10.2 点云引擎 (Point Cloud Engine) .....	367
5.11 分析 (Analysis) .....	368
5.11.1 能量数据 (Energy Data) .....	368
5.11.2 分析可视化 (Analysis Visualization) .....	369
5.11.3 概念能量分析 (Conceptual Energy Analysis) .....	373
5.11.4 能量分析详细模型 (Detailed Energy Analysis Model) .....	374
5.12 地点和位置 (Place and Locations) .....	377
5.13 工作共享 (Worksharing) .....	381
5.13.1 工作集中的图元 (Elements in Worksets) .....	381
5.13.2 图元所有权 (Element Ownership) .....	383
5.13.3 打开工作共享文件 (Opening a Workshared Document) .....	383
5.13.4 可见性和显示 (Visibility and Display) .....	384
5.13.5 工作集 (Worksets) .....	388
5.13.6 工作共享文件管理 (Workshared File Management) .....	389
5.14 构造建模 (Construction Modeling) .....	391
5.14.1 部件和视图 (Assemblies and Views) .....	391
5.14.2 零件 (Parts) .....	392
5.15 链接文件 (Linked Files) .....	394
5.15.1 Revit 链接 (Revit Links) .....	394
5.15.2 管理外部文件 (Managing External Files) .....	399
5.16 导出 (Export) .....	402
5.16.1 导出表 (Export Tables) .....	405
5.16.2 导出 IFC (IFC Export) .....	406
5.16.3 自定义导出 (Custom Export) .....	407
附录 A 术语 (Glossary) .....	408
附录 B 疑问解答 (FAQ) .....	410
B.1 常见问题 (General Questions) .....	410
B.2 Revit Structure 问题 (Revit Structure Questions) .....	411
附录 C VB.NET 代码的 “Hello World” .....	413
C.1 创建新项目 (Create a New Project) .....	413



C.2	添加引用和命名空间 (Add Reference and Namespace)	413
C.3	更改类名 (Change the Class Name)	414
C.4	添加代码 (Add Code)	415
C.5	创建.addin 清单文件 (Create a .addin Manifest File)	415
C.6	生成程序 (Build the Program)	416
C.7	调试程序 (Debug the Program)	416
附录 D	内部单元的材料属性 (Material Properties Internal Units)	418
附录 E	API 用户界面指南 (API User Interface Guidelines)	421
E.1	引言 (Introduction)	421
E.2	一致性 (Consistency)	421
E.3	使用用户语言 (Speak the Users' Language)	421
E.4	高质量布局 (Good Layout)	421
E.5	设好默认值 (Good Defaults)	421
E.6	渐进式展开 (Progressive Disclosure)	422
E.7	本地化用户界面 (Localization of the User Interface)	422
E.8	对话框指南 (Dialog Guidelines)	423
E.9	功能区指南 (Ribbon Guidelines)	452
E.10	通用定义 (Common Definitions)	458
E.11	用语定义 (Terminology Definitions)	459

# 第 1 章 引言 (Introduction)

## 1.1 欢迎使用 Revit 平台 API (Welcome to the Revit Platform API)

基于 Revit 的所有产品都是参数化的 BIM 工具。这些工具类似于 CAD 程序，但除了用于二维制图外，还用于三维建模。在 Revit 中，柱和墙等真实图元可以置于模型中。建立模型之后，就能创建诸如剖面、图例等模型视图，这些视图是由三维物理模型生成的。因而，在某个视图中所作的修改会自动传送到所有视图中。当修改模型时，几乎不需要去更新多处图纸和一些细节。

### 1.1.1 Revit 平台 API 简介 (Introduction to the Revit Platform API)

Revit .NET API 可用 Visual Basic.NET (以下简称“VB.NET”)、C#和 C++/CLI 等任何与.NET 兼容的编程语言进行编程。

Revit Architecture 2013、Revit Structure 2013 和 Revit MEP 2013 均包含 Revit 平台 API，以便将应用程序与 Revit 集成一体。这三个 API 非常相似，共称为 Revit 平台 API。使用 API 之前，学会使用 Revit 并了解其产品特点，能更好地理解应用程序编程相关的知识。学用 Revit 有以下帮助：

- 保持与 Revit 用户界面及其命令的一致性。
- 设计出无缝插件应用程序。
- 有效掌握 API 类及其类成员运用。

若不熟悉 Revit 或 BIM，可从 [www.autodesk.com/revit](http://www.autodesk.com/revit) 网站 Revit 产品中心获得更多知识。

### 1.1.2 Revit 平台 API 能做什么 (What Can You Do with the Revit Platform API)

Revit 平台 API 可以：

- 访问模型的图形数据。
- 访问模型的参数数据。
- 创建、编辑、删除诸如楼层、墙、柱等模型图元。
- 创建自动执行重复任务的插件。
- 将应用程序集成到基于 Revit 的各层产品。例如，将外部关系数据库连接到 Revit，或发送模型数据至分析程序。
- 完成所有 BIM 应用过程中的所有分析。
- 自动创建项目文件。



### 1.1.3 必要条件 (Requirements)

读懂并掌握本书所述, 需具备下列预备知识和技能:

- (1) 了解 Revit Architecture 2013、Revit Structure 2013 或 Revit MEP 2013。
- (2) 熟悉 C#或 VB.NET 等某一门兼容编程语言的通用语言规范。
- (3) 熟悉 Microsoft Visual Studio 2010 或 Microsoft Visual Studio 2010 Express 版开发环境, 也可以使用 Revit 内置的 Sharp Develop 开发环境。
- (4) 熟悉 Microsoft .NET Framework 4.0。
- (5) 从欧特克开发人员网站下载或在 Revit 安装盘上找出 (<DVD\_Drive>: \Utilities\ Common\Software Development Kit) Revit 软件开发工具包 (SDK)。

### 1.1.4 安装 (Installation)

与 Revit Architecture、Revit Structure 和 Revit MEP 一起安装 Revit 平台 API。基于 .NET 的任何应用程序都引用 Revit 程序目录中的 RevitAPI.dll 和 RevitAPIUI.dll。RevitAPI.dll 包含用于访问 Revit 应用程序、文件、图元和参数等的数据库级方法, 而 RevitAPIUI.dll 包含操控、定制 Revit 用户界面相关的接口。

### 1.1.5 受支持的编程语言 (Supported Programming Languages)

Revit 平台 API 完全支持 Microsoft .NET Framework 4.0 兼容编程语言, 诸如 VB.NET 或 Visual C#语言等。

### 1.1.6 使用手册 (User Manual)

本书是 Revit 软件开发工具包 (SDK) 的一部分, 介绍了如何使用 Revit 平台 API 实现插件应用程序。

在创建 Revit 插件应用程序之前, 请通读本书并尝试使用书中的示例代码。已经具备一定的 Revit 平台 API 使用经验的开发人员, 也许只需查看一下“注意事项”和“排错”部分的内容。

#### 1. Revit 平台 API 介绍 (Introduction to the Revit Platform API)

欢迎使用 Revit 平台 API——介绍了 Revit 平台 API 和首次创建插件之前的一些必备知识。

入门——循序渐进地说明了如何用 Visual Studio 2010 创建首个 Revit 插件应用程序“Hello World”, 以及其他四个包括基本插件函数的程序代码的演练。

#### 2. 基础专题 (Basic Topics)

这些章节涵盖了 Revit 平台 API 的基本功能机理。

插件集成——讨论了怎样将插件集成到 Revit 用户界面, 如何通过用户指令或特定 Revit 事件如程序启动来调用。

应用程序和文件——应用程序和文件类分别表示 Revit 平台 API 中的应用程序和项目文件, 本章节介绍了一些基本概念及相关章节的概况。

图元概要——Revit 项目中的大部分数据都存放于图元集合中。本章讨论了基本的图元



机理、特性和分类。

过滤——用于从文件中获取一组图元。

选集——某个文件中所选操作图元的集合。

参数——大部分图元信息存储在参数中，本章讨论了参数功能。

集合——一些实用集合类型，诸如数组、映射、设置集合和相关的迭代器等。

### 3. 图元专题 (Element Topics)

基于图元分类来介绍图元。在了解单个图元之前，确信已理解图元概要和参数章节。

编辑图元——了解怎样移动、旋转、删除、镜像、分组、阵列图元。

墙、楼板、屋顶和洞口——讨论了一些图元，其对应的图元类型表示 API 中内建的建筑类型和各种洞口类型。

族实例——了解族和族实例之间的关系、族和族实例的特性，以及它们如何载入或创建。

族创建——了解如何创建和修改 Revit 族文件。

概念设计——讨论了如何在 Revit 概念体量文件中创建复杂的几何结构和形状。

基准和信息图元——了解如何建立轴网、添加标高、运用设计选项等。

注释图元——讨论了文件注释，包括添加尺寸、详图曲线、标记和注释符号。

草图——草图功能包括二维和三维草图类，如 SketchPlane、ModelCurve、Generic Form 等。

视图——了解观察模型和构件的各种方式，以及如何在 API 中操控视图。

材料——材料数据是一个用来识别项目所用物理材料和质地、颜色等属性的图元。

### 4. 进阶专题 (Advanced Topics)

几何结构——讨论了 API 中与图形相关的类型，用于描述模型的图形表示，包括描述和储存几何信息的三个类。

地点和位置——定义项目位置，包括城市、国家、经纬度。

共享参数——共享参数是内含参数说明的外部文本文件。本章介绍了如何通过 Revit 平台 API 访问共享参数。

事务——介绍了事务的两个用途和使用事务时必须考虑的限制。

事件——讨论了如何利用 Revit 事件。

动态模型更新——了解如何运用更新器修改模型以反映文件中的更改。

故障发布及处理——了解如何发布故障，并运用交互式 Revit 故障处理机制。

分析可视化——了解如何显示 Revit 项目中的分析成果。

### 5. 专业产品 (Product Specific)

Revit 产品包括 Revit Architecture、Revit Structure 和 Revit MEP。某些 API 只适用于某专业产品。

Revit Architecture——讨论了特定用于 Revit Architecture 的 API。

Revit Structure——讨论了特定用于 Revit Structure 的 API。

Revit MEP——讨论了特定用于 Revit MEP 的 API。



## 6. 其他 (Other)

术语表——定义了一些用于本书的术语。

附录——一些附加信息资料，如常见问题解答，如何运用 VB.NET 编程，等等。

### 1.1.7 文档协定 (Documentation Conventions)

本书包含命名空间格式中的类名称，如 Autodesk.Revit.DB.Element。在 C++/CLI 中，Autodesk.Revit.Element 为 Autodesk:: Revit:: DB:: Element。因本书中的示例代码均为 C#编写，故默认的命名空间为 Autodesk.Revit.Element。如果想要看 Visual Basic 代码编写的示例，可在 SDK 示例目录中找到一些 VB.NET 应用程序。

某些 Revit 平台 API 类属性是“索引型”的，在 API 帮助文件 (RevitAPI.chm) 中也称之为重载。例如，Element.Geometry 属性，本书中称之为属性，尽管在 C#代码中可作为方法访问它们，但要在属性名称前加上前缀“get\_”或“set\_”。例如，为了使用 Element.Geometry (选项) 属性，应该用 Element.get\_Geometry (选项) 获取该属性。

### 1.1.8 本版本的新特性

有关版本变更和新特性的信息，请参见 Revit 2014 API.chm 中“新特性”部分。

## 1.2 入门 (Getting Started)

任何微软 .NET Framework 4.0 兼容编程语言，如 Visual C#或 VB.NET，都可完全访问 Revit 平台 API。Visual C#和 VB.NET 都是开发 Revit 平台 API 应用程序的常用语言。但本书的重点是使用 Visual C#来开发应用程序。

### 1.2.1 演练 (Walkthroughs)

对于 Revit 平台 API 新手，熟悉以下专题是了解产品的良好开端。这些演练为一些常见方案提供了循序渐进式的指导，以帮助开发人员了解该产品或某个特定功能。以下演练将帮助开发人员上手使用 Revit 平台 API。

演练：Hello World——演示了如何用 Revit 平台 API 创建一个插件。

演练：Add Hello World Ribbon Panel——演示了如何添加一个自定义功能区面板。

演练：Retrieve Selected Elements——演示了如何检索选中的图元。

演练：Retrieve Filtered Elements——演示了如何根据过滤条件检索图元。

### 1.2.2 演练：Hello World (Walkthrough: Hello World)

根据提供的指导，用 Revit 平台 API 和 C#创建 Hello World 程序。关于如何用 VB.NET 创建插件应用程序，请参阅附录 C：VB.NET 编写的 Hello World 程序。

Hello World 演练涵盖了以下主题：

- 新建项目。
- 添加引用。
- 变更类名。
- 编写代码。



- 调试插件。

本节中的所有操作和代码均使用 Visual Studio 2010 创建。

### 1. 新建项目 (Create a New Project)

Visual Studio 编写 C# 程序的第一步是选择一种项目类型并创建新的类库。

(1) 从“文件”菜单选择**新建** ➤ **项目...**。

(2) 从已安装的样板框架中，单击 **Visual C#**。

(3) 在右边框架内，单击**类库** (图 1-1，此演练假定项目位置是 D: \Sample)。

(4) 在名称字段，键入“Hello World”作为项目名称。

(5) 单击**确定**。

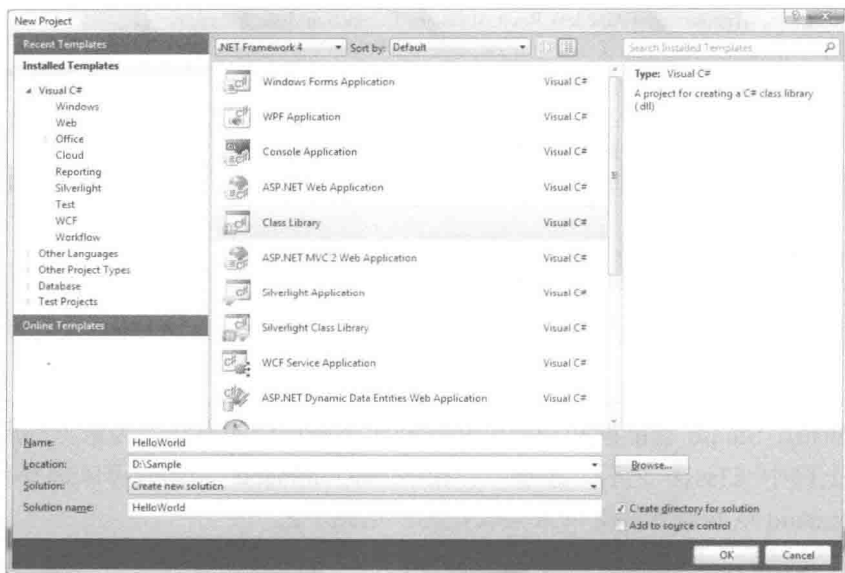


图 1-1 添加新项目

### 2. 添加引用 (Add References)

(1) 添加 Revit 引用。

- 如果解决方案资源管理器窗口未打开，则从“视图”菜单中选择解决方案资源管理器。
- 在解决方案资源管理器中，右键单击**引用**以显示上下文菜单。
- 从上下文菜单中，单击**添加引用**。出现“添加引用”对话框。
- 在添加引用对话框中，单击**浏览**选项卡。找到 Revit 安装的文件夹，然后单击 RevitAPI.dll。例如，安装文件夹的位置通常是 C: \Program Files\Autodesk\Revit Architecture 2012\Program\RevitAPI.dll。
- 单击**确定**以选择 .dll 并关闭对话框。RevitAPI 会出现在解决方案资源管理器引用树中。
- 请注意，对于新项目，RevitAPI 的“复制本地”属性应始终设置为 false。这会节省硬盘空间，并避免 Visual Studio 调试器不知道使用哪个 DLL 副本。右键单





击 RevitAPI.dll, 选择属性, 并将“复制本地”设置从 true (默认设置) 改为 false。

(2) 对 RevitAPIUI.dll 重复上述步骤。

### 3. 添加代码 (Add Code)

添加代码 1-1 以创建插件。

#### 代码 1-1: Getting Started

```
using System;

using Autodesk.Revit.UI;
using Autodesk.Revit.DB;
namespace HelloWorld
[Autodesk.Revit.Attributes.Transaction (Autodesk.Revit.Attributes.TransactionMode.Automatic)]
public class Class1 : IExternalCommand
{
    {
        public Autodesk.Revit.UI.Result Execute (ExternalCommandData revit,
            ref string message, ElementSet elements)
        {
            TaskDialog.Show ("Revit", "Hello World");
            return Autodesk.Revit.UI.Result.Succeeded;
        }
    }
}
```

提示: Visual Studio 智能感知功能可帮助创建实现接口的方法, 为所需的所有方法添加存根。在上例中 Class1 之后, 添加 “: IExternalCommand”, 此后可以选择 Implement IExternalCommand 从智能感知菜单来获取代码, 见图 1-2。

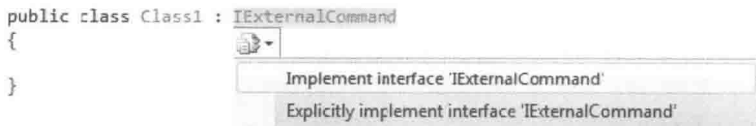


图 1-2 用智能感知功能实现接口

每个 Revit 插件应用程序都必须有一个进入点类来实现 IExternalCommand 接口, 且必须实现 Execute() 方法。Execute() 方法是插件程序的进入点, 类似于其他程序中的 Main() 方法。插件程序进入点类的定义包含在一个程序集内。有关详细信息, 请参阅 1.3 节插件集成。

### 4. 构建程序 (Build the Program)

在完成代码之后, 必须构建文件。从“构建”菜单单击生成解决方案。若构建输出出现在输出窗口中, 则说明项目编译已成功。

### 5. 创建.addin 清单文件 (Create a .addin Manifest File)

HelloWorld.dll 文件出现在项目输出目录中。在 Revit 中调用该应用程序, 则需创建一个清单文件将它注册到 Revit。



- (1) 要创建清单文件, 可用文本编辑器新建一个文本文件。
- (2) 添加代码 1-2 所示文本。

#### 代码 1-2: 为外部命令创建 .addin 清单文件

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<RevitAddIns>
  <AddIn Type="Command">
    <Assembly>D: \Sample\HelloWorld\bin\Debug\HelloWorld.dll</Assembly>
    <AddInId>239BD853-36E4-461f-9171-C5ACEDA4E721</AddInId>
    <FullClassName>HelloWorld.Class1</FullClassName>
    <Text>HelloWorld</Text>
    <VendorId>ADSK</VendorId>
    <VendorDescription>Autodesk, www.autodesk.com</VendorDescription>
  </AddIn>
</RevitAddIns>
```

(3) 以 HelloWorld.addin 为文件名并保存到以下位置:

- 对于 Windows XP 系统, 保存到 C: \Documents and Settings\All Users\Application Data\Autodesk\Revit\Addins\2012\。
- 对于 Vista/Windows 7 系统, 保存到 C: \ProgramData\Autodesk\Revit\Addins\2012\。
- 如果所用的应用程序集 dll 是在网络共享而不是在本地硬盘上, 则必须修改 Revit.exe.config, 以允许加载本地计算机以外的 .NET 程序集。在 Revit.exe.config 的 “runtime” 节点, 添加要素 <loadFromRemoteSources enabled="true"/>, 如下所示。

```
<runtime>
  <generatePublisherEvidence enabled="false" />
  <loadFromRemoteSources enabled="true"/>
</runtime>
```

有关使用清单文件的详细信息, 请参阅 1.3 节插件集成。

#### 6. 调试插件 (Debug the Add-in)

在调试模式下运行程序可使用断点暂停程序, 以便检查变量和对象的状态。如果出现错误, 则可以检查程序运行的变量, 来推断为什么不是预期的值。

- (1) 在解决方案资源管理器窗口, 右键单击 HelloWorld 项目以显示上下文菜单。
- (2) 从上下文菜单单击 **Properties**, 显示 “属性” 窗口。
- (3) 单击 **Debug** 选项卡。
- (4) 在 “Start Action” 区段下方, 单击 **Start external program** 浏览 Revit.exe 文件, 见图 1-3。默认情况下, 该文件位于 C: \Program Files\Autodesk\Revit Structure 2012\Program\Revit.exe。

- (5) 从 “调试” 菜单中, 选择切换断点或按 F9 键切换到下面的行上设置断点。

```
TaskDialog.Show ("Revit", "Hello World");
```

- (6) 按 F5 键启动调试程序。测试调试过程如下:

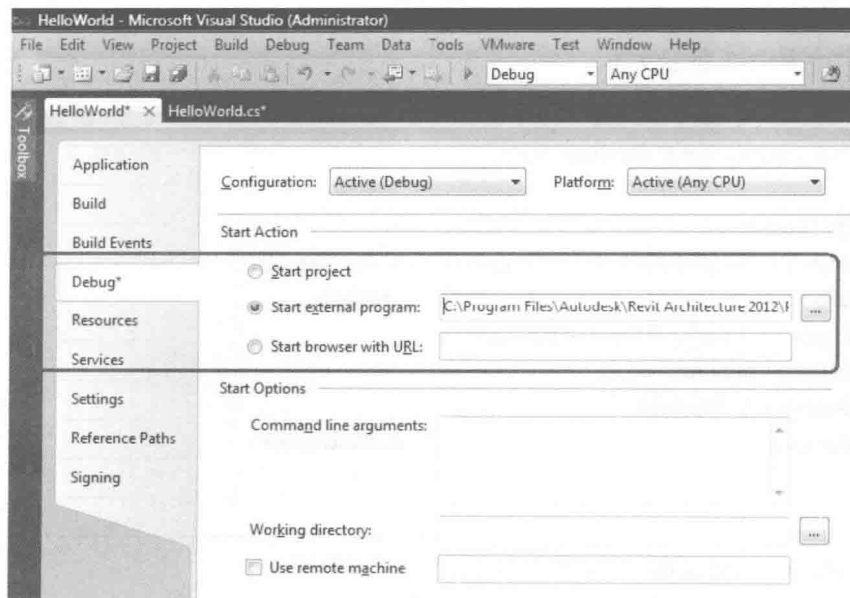


图 1-3 设置调试环境

- 在插件选项卡上,如图 1-4 所示 **HelloWorld** 按钮出现在 External Tools 菜单按钮上。

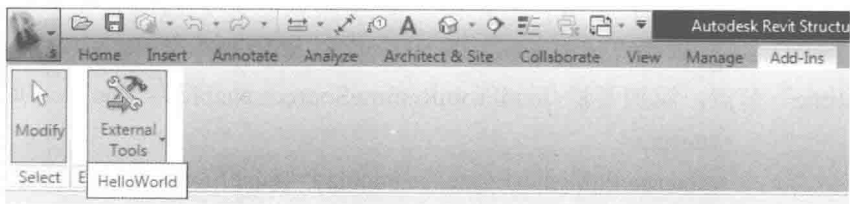


图 1-4 HelloWorld 外部工具命令

- 单击 **HelloWorld** 执行程序,激活断点。
- 按 F5 键继续执行程序,显示图 1-5 所示系统消息。

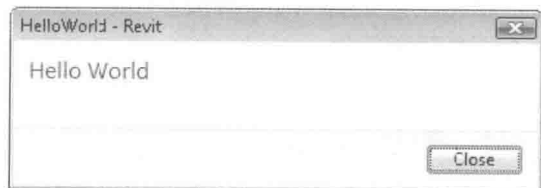


图 1-5 任务对话框消息

问: 为什么看不到插件选项卡或为什么我的插件应用程序未显示在“External Tools”菜单按钮下方?

答: 在多数情况下,如果插件应用程序加载失败,Revit 在启动时将显示一个错误对话框提示有关故障信息。例如,如果在清单文件中指定的位置无法找到该插件 DLL,会显示类似图 1-6 所示消息。

## 7. 故障排除 (Troubleshooting)

问: 我的插件应用程序无法编译。

答: 当编译示例代码时,如果出现错误,则该问题可能与用来编译插件程序的 RevitAPI 版本有关。删除旧的 RevitAPI 引用并加载一个新的引用。有关详细信息,请参阅前文“添加引用”章节。



如果在 `ECClassName` 中指定的类名称找不到, 或并非继承自 `IExternalCommand`, 也将显示错误消息。

然而, 在某些情况下, 加载插件程序失败可能不显示任何消息。可能的原因包括:

- 使用了不同版本的 RevitAPI 编译插件应用程序。
- 未找到清单文件。
- .addin 清单文件中存在格式错误。

问: 为什么我的插件应用程序无法运行?

答: 即使插件程序显示在 “External Tools” 菜单按钮下方, 它也可能无法正常运行。这通常是由于代码异常, 如代码 1-3。

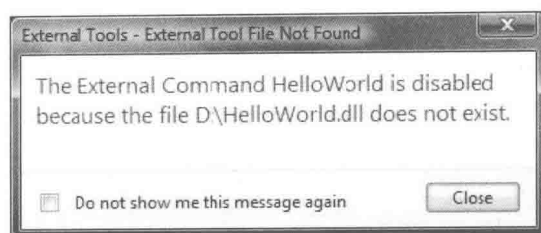


图 1-6 外部工具错误消息

### 代码 1-3: Exceptions in Execute()

```
Command: IExternalCommand
{
    A a = new A(); //line x
    public IExternalCommand.Result Execute()
    {
        //...
    }
}
Class A
{
    //...
}
```

以下两个异常清楚地说明了问题:

- 在 line x 行有错。
- `Execute()` 方法将引发异常。

在命令失败时, Revit 将显示一个如图 1-7 所示的包含未捕获异常信息的错误对话框。

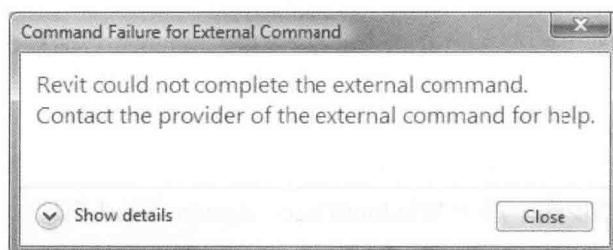


图 1-7 外部命令未处理的异常

这可用于辅助调试运行的命令。部署给用户的命令应该在该示例入口方法中运用 “try.. catch.. finally”, 以免 Revit 捕获异常。请看代码 1-4。

**代码 1-4: 在程序执行中使用 try catch**

```
public IExternalCommand.Result Execute (ExternalCommandData commandData, ref string message, ElementSet elements)
{
    ExternalCommandData cdata = commandData;
    Autodesk.Revit.ApplicationServices.Application app = cdata.Application;

    try
    {
        // Do some stuff
    }

    catch (Exception ex)
    {
        message = ex.Message;
        return Autodesk.Revit.UI.Result.Failed;
    }

    return Autodesk.Revit.UI.Result.Succeeded;
}
```

**1.2.3 演练: 添加 Hello World 功能区面板 (Add Hello World Ribbon Panel)**

前面一节介绍了如何创建插件应用程序并在 Revit 中调用它。还了解了创建addin 清单文件将插件应用程序注册为一个外部工具。Revit 调用插件程序的另一种方法是通过自定义功能区面板。

**1. 新建项目 (Create a New Project)**

完成以下步骤来创建一个新项目:

- (1) 在 Visual Studio 中使用类库样板创建 C# 项目。
- (2) 键入 AddPanel 作为项目名称。
- (3) 按前一个演练(演练: Hello World)中的说明, 添加对 RevitAPI.dll 和 RevitAPIUI.dll 的引用。

**(4) 添加 PresentationCore 参考 (图 1-8)。**

- 在解决方案资源管理器中, 右键单击 **References** 以显示上下文菜单。
- 从上下文菜单中, 单击 **Add Reference**, Add Reference 对话框出现。
- 在 Add Reference 对话框中, 单击.NET 选项卡。
- 从组件名称列表中, 选择 **PresentationCore**。
- 单击 **OK** 关闭对话框, **PresentationCore** 出现在解决方案资源管理器引用树。
- (5) 遵循上述类似步骤, 添加 WindowsBase、System.Xaml 引用。

**2. 更改类名 (Change the Class Name)**

要更改类名称, 请完成以下步骤:

- (1) 在类视图窗口中, 右键单击 Class1 以显示上下文菜单。
- (2) 从上下文菜单中, 选择 **Rename** 并更改类名为 CsAddPanel。
- (3) 在解决方案资源管理器中, 右键单击 Class1.cs 文件来显示上下文菜单。

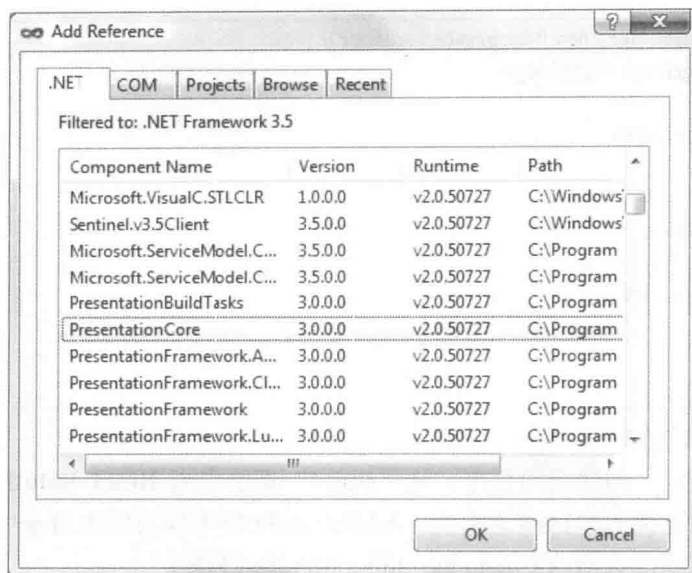


图 1-8 添加引用

(4) 从上下文菜单中, 选择 **Rename** 并更改文件名为 **CsAddPanel.cs**。

(5) 双击 **CsAddPanel.cs** 将其打开以进行编辑。

### 3. 添加代码 (Add Code)

添加面板项目不同于 **Hello World** 项目, 因为在 **Revit** 运行时它是自动调用的。此项目使用 **IEternalApplication** 接口。**IEternalApplication** 接口包含两个抽象方法, 即 **OnStartup()** 和 **OnShutdown()**。关于 **IEternalApplication** 的更多信息, 请参见 1.3 节插件集成。功能区添加面板见代码 1-5。

#### 代码 1-5: Adding a ribbon panel

```
using Autodesk.Revit.UI;
using Autodesk.Revit.DB;
using System.Windows.Media.Imaging;
class CsAddpanel : Autodesk.Revit.UI.IEternalApplication
{
    public Autodesk.Revit.UI.Result OnStartup (UIControlledApplication application)
    {
        // add new ribbon panel
        RibbonPanel ribbonPanel = application.CreateRibbonPanel ("NewRibbonPanel");

        //Create a push button in the ribbon panel "NewRibbonPanel"
        //the add-in application "HelloWorld" will be triggered when button is pushed

        PushButton pushButton = ribbonPanel.AddItem (new PushButtonData ("HelloWorld",
            "HelloWorld", @"D: \HelloWorld.dll", "HelloWorld.CsHelloWorld")) as PushButton;

        // Set the large image shown on button
        Uri uriImage = new Uri (@@"D: \Sample\HelloWorld\bin\Debug\39-Globe_32x32.png");
```





```
BitmapImage largeImage = new BitmapImage (uriImage);
pushButton.LargeImage = largeImage;

return Result.Succeeded;
}

public Result OnShutdown (UIControlledApplication application)
{
    return Result.Succeeded;
}
}
```

#### 4. 构建程序 (Build the Application)

在完成代码之后, 构建应用程序。从“Build”菜单单击 **Build Solution**。若构建输出出现在输出窗口, 则说明项目编译成功。AddPanel.dll 位于项目输出目录中。

#### 5. 创建.addin 清单文件 (Create the .addin Manifest File)

如果在 Revit 中调用应用程序, 则需创建清单文件注册到 Revit。

(1) 用文本编辑器创建文本文件。

(2) 在文件中添加代码 1-6 文本。

#### 代码 1-6: Creating a .addin file for an external application

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<RevitAddIns>
<AddIn Type="Application">
  <Name>SampleApplication</Name>
  <Assembly>D: \Sample\AddPanel\AddPanel\bin\Debug\AddPanel.dll</Assembly>
  <AddInId>604B1052-F742-4951-8576-C261D1993107</AddInId>
  <FullClassName>AddPanel.CsAddPanel</FullClassName>
  <VendorId>ADSK</VendorId>
  <VendorDescription>Autodesk, www.autodesk.com</VendorDescription>
</AddIn>
</RevitAddIns>
```

(3) 以 HelloWorldRibbon.addin 为文件名保存到以下位置:

- 对于 Windows XP 系统, 保存到 C: \Documents and Settings\All Users\Application Data\Autodesk\Revit\Addins\2012\。
- 对于 Vista/Windows 7 系统, 保存到 C: \ProgramData\Autodesk\Revit\Addins\2012\。

注: AddPanel.dll 文件在一个默认名为 Debug 的新文件夹中 (D: \Sample\HelloWorld\bin\Debug\AddPanel.dll)。使用此文件路径作为程序集的位置。

有关详细信息, 请参阅 1.3 节插件集成。

#### 6. 调试 (Debugging)

启动调试, 生成项目, 然后运行 Revit。如图 1-9 所示一个新的 NewRibbonPanel 功能面板显示在插件选项卡上, 带有地球仪大图标 Hello World 是面板上唯一的按钮。

单击 Hello World 运行程序会显示如图 1-10 所示对话框。

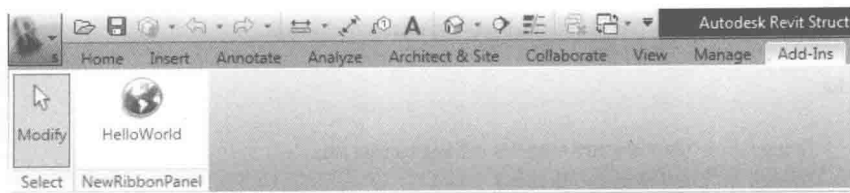


图 1-9 将新功能区面板添加到 Revit

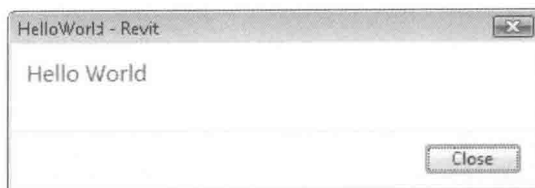


图 1-10 HelloWorld 对话框

#### 1.2.4 演练：检索所选图元 (Retrieve Selected Elements)

本节介绍从 Revit 获取所选图元的插件应用程序。

在插件应用程序中，可以对特定图元进行特定操作。例如，可以获取或更改某图元的参数值。完成以下步骤以获取参数值。

(1) 如同先前的演练程序所述，创建一个新项目并添加引用。

(2) 使用 `UIApplication.ActiveUIDocument.Selection.Elements` 属性来检索所选对象。

所选对象为 `Revit SelElementSet`。使用 `IEnumerator` 接口或 `foreach` 循环来搜索 `ElementSet`。

代码 1-7 说明了如何检索所选定的图元。

##### 代码 1-7: Retrieving selected elements

```
[Autodesk.Revit.Attributes.Transaction (TransactionMode.ReadOnly)]
public class Document_Selection : IExternalCommand
{
    public Autodesk.Revit.UI.Result Execute (ExternalCommandData commandData,
        ref string message, ElementSet elements)
    {
        try
        {
            // Select some elements in Revit before invoking this command

            // Get the handle of current document.
            UIDocument uidoc = commandData.Application.ActiveUIDocument;
            // Get the element selection of current document.
            Selection selection = uidoc.Selection;
            ElementSet collection = selection.Elements;

            if (0 == collection.Size)
            {
                // If no elements selected.
```



```
        TaskDialog.Show ("Revit", "You haven't selected any elements.");
    }
    else
    {
        String info = "Ids of selected elements in the document are: ";
        foreach (Element elem in collection)
        {
            info += "\n\t" + elem.Id.IntegerValue;
        }

        TaskDialog.Show ("Revit", info);
    }
}
catch (Exception e)
{
    message = e.Message;
    return Autodesk.Revit.UI.Result.Failed;
}

return Autodesk.Revit.UI.Result.Succeeded;
}
```

在获得所选图元之后，即可获取所选图元的属性或参数。有关更多信息，请参见 2.3 节参数。

### 1.2.5 演练：检索过滤图元 (Retrieve Filtered Elements)

使用过滤器可以筛选出满足某一条件的图元。有关创建和使用图元过滤器的更多信息，请参阅 1.5.3 节中的过滤图元集。

代码 1-8 检索文件中的所有门并显示一个对话框，列出所有门的 ID。

#### 代码 1-8: Retrieve filtered elements

```
// Create a Filter to get all the doors in the document
ElementClassFilter familyInstanceFilter = new ElementClassFilter (typeof (FamilyInstance));
ElementCategoryFilter doorsCategoryfilter = new ElementCategoryFilter (BuiltInCategory.OST_Doors);
LogicalAndFilter doorInstancesFilter =
    new LogicalAndFilter (familyInstanceFilter, doorsCategoryfilter);
FilteredElementCollector collector = new FilteredElementCollector (document);
ICollection<ElementId> doors = collector.WherePasses (doorInstancesFilter).ToElementIds();

String prompt = "The ids of the doors in the current document are: ";
foreach (ElementId id in doors)
{
    prompt += "\n\t" + id.IntegerValue;
}

// Give the user some information
TaskDialog.Show ("Revit", prompt);
```



## 1.3 插件集成 (Add-in integration)

开发人员可通过创建和实现外部命令及外部应用程序来添加程序功能。Revit 会使用.addin 清单文件来识别外部命令和应用程序。

- 外部命令显示在插件选项卡上的外部工具菜单按钮下方。当 Revit 启动时会调用外部应用程序，而 Revit 关闭时会自动卸载。

本章重点介绍以下方面内容：

- 了解如何使用外部命令和外部应用程序添加功能。
- 如何访问 Revit 事件。
- 如何自定义 Revit 用户界面。

### 1.3.1 概述 (Overview)

Revit 平台 API 是基于 Revit 应用程序功能的，由两个只在 Revit 运行期间才运行的类库组成。

RevitAPI.dll 包含用于访问 Revit 的应用程序、文件、图元和参数的数据库级的方法。它还包含 IExternalDBApplication 及相关接口。

RevitAPIUI.dll 包含所有与操控和定制 Revit 用户界面相关的 API 接口，包括：

- IExternalCommand 及相关接口。
- IExternalApplication 及相关接口。
- Selection。
- RibbonPanel、RibbonItem 及其子类。
- TaskDialogs。

如图 1-11 所示，Revit Architecture、Revit Structure 和 Revit MEP 分别对应建筑、结构和设备。

要创建一个基于 RevitAPI 的插件，必须在插件 DLL 中提供特定的进入点类型。这些进入点的类实现接口，IExternalCommand、IExternalApplication 两者之一，或 IExternalDBApplication。这样，会以某些事件的方式自动运行插件，或者在用 IExternalCommand 和 IExternalApplication 的情况下，从外部工具菜单按钮手工启动。

IExternalCommand、IExternalApplication、IExternalDBApplication 和其他用于插件集成的 Revit 事件，均在本章一并介绍。

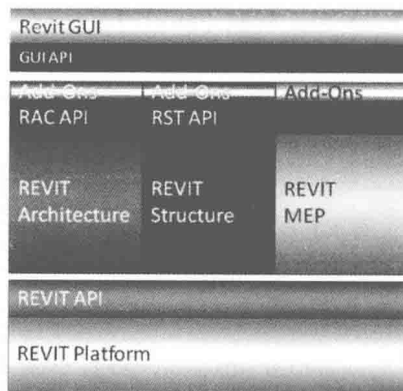


图 1-11 Revit、RevitAPI 及插件

### 1.3.2 外部命令 (External Commands)

开发人员可以通过实现外部工具菜单按钮上的外部命令来添加功能。

#### 1. 加载和运行外部命令 (Loading and Running External Commands)

当 Revit 没有其他活动的命令或编辑模式时，可以启用已注册的外部命令。当选中一个命令时，即创建了一个命令对象并调用其 Execute() 方法。而一旦此方法返回 Revit，命



令对象即被销毁。因此，命令执行之间的对象数据无法存留。不过，有其他方法可以保存命令执行之间的数据。例如，可以使用 Revit 共享参数机制来存储 Revit 项目数据。

外部命令可以添加到外部工具菜单按钮下方的外部工具面板上，也可作为插件选项卡、分析选项卡或新的自定义功能区选项卡上的自定义功能区面板。这两种方法的例子，请参阅 1.2.2 节演练：Hello World 和 1.2.3 节演练：Add Hello World Ribbon Panel。

外部工具、功能区选项卡和功能区面板在启动时会被初始化。初始化步骤如下：

- Revit 读取清单文件并识别：
  - 可被调用的外部应用程序。
  - 可被添加到 Revit 外部工具菜单按钮的外部工具。
- 外部应用程序会将面板和内容添加到插件选项卡。

## 2. 外部命令接口 (IExternalCommand)

通过创建一个实现 IExternalCommand 接口的对象来创建外部命令。IExternalCommand 接口有一个抽象方法 Execute(), 这是外部命令的主要方法。

Execute() 有三个参数：

- commandData (ExternalCommandData)。
- message (String)。
- elements (ElementSet)。

(1) commandData (ExternalCommandData)。ExternalCommandData 对象包含引用的应用程序和外部命令所需视图。外部命令从此参数直接或间接检索所有 Revit 数据。

例如，代码 1-9 说明了如何从 commandData 参数中检索 Autodesk.Revit. Document。

### 代码 1-9：检索活动文件

```
Document doc = commandData.Application.ActiveUIDocument.Document;
```

表 1-1 说明了 ExternalCommandData 公共属性。

表 1-1 ExternalCommandData 公共属性

属 性	说 明
Application (Autodesk.Revit.UI.UIApplication)	检索一个表示当前外部命令 UIApplication 的对象
JournalData (IDictionary<String, String>)	用于读写 Revit 日志文件数据的数据映射
View (Autodesk.Revit.DB.View)	检索一个代表外部命令工作视图的对象

(2) message (String)。外部命令使用输出参数返回错误信息。字符串类型的参数在外部命令过程中设置。当设置了消息参数，在返回 Autodesk.Revit.UI.Result.Failed 或 Autodesk.Revit.UI.Result.Cancelled 时，将显示一个错误消息对话框。

代码 1-10 阐释了如何使用消息参数。

### 代码 1-10：设置一个错误信息字符串

```
class IExternalCommand_message : IExternalCommand
{
    public Autodesk.Revit.UI.Result Execute (
```



```
Autodesk.Revit.ExternalCommandData commandData, ref string message,
Autodesk.Revit.ElementSet elements)
{
    message = "Could not locate walls for analysis.";
    return Autodesk.Revit.UI.Result.Failed;
}
}
```

执行上述外部命令会导致出现如图 1-12 所示对话框。

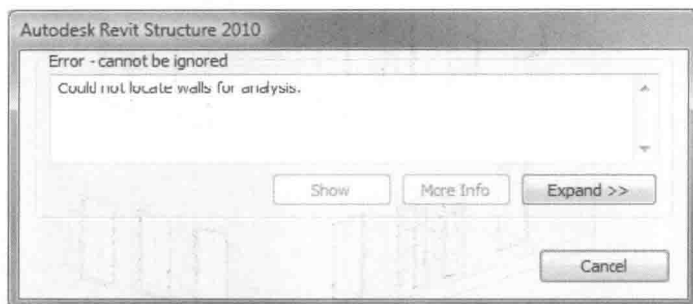


图 1-12 错误消息对话框

(3) elements (ElementSet)。每当返回 Autodesk.Revit.UI.Result.Failed 或 Autodesk.Revit.UI.Result.Canceled 且参数消息不为空时，都会出现一个错误或警告对话框。此外，如果任何图元被添加到图元参数，则这些图元都将突出显示在屏幕上。无论命令是否失败，也不论图元是否返回，设置消息参数是个好的编程习惯。

代码 1-11 突出显示了预先选定的墙。

#### 代码 1-11: Highlighting walls

```
class IExternalcommand_elements : IExternalCommand
{
    public Result Execute (
        Autodesk.Revit.UI.ExternalCommandData commandData, ref string message,
        Autodesk.Revit.DB.ElementSet elements)
    {
        message = "Please note the highlighted Walls.";
        FilteredElementCollector collector =
        new FilteredElementCollector (commandData.Application.ActiveUIDocument.Document);
        ICollection<Element> collection = collector.OfClass (typeof (Wall)).ToElements();
        foreach (Element e in collection)
        {
            elements.Insert (e);
        }

        return Result.Failed;
    }
}
```



图 1-13 显示了代码 1-11 的结果。

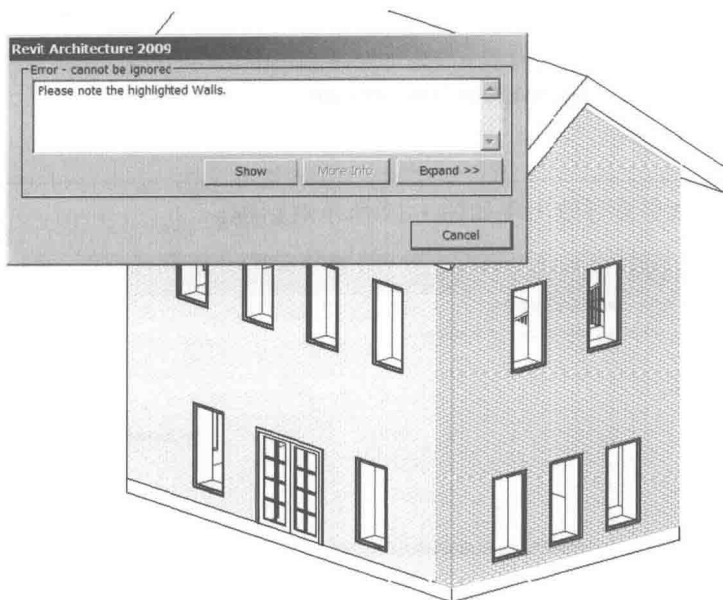


图 1-13 错误消息对话框和突出显示的墙

(4) 返回 (Return)。返回结果指明命令执行失败、成功或是已被用户取消，见表 1-2。若未成功，则 Revit 会逆转外部命令所作的更改。

表 1-2

IExternalCommand.Result

成员名称	说 明
Autodesk.Revit.UI.Result.Succeeded	外部命令成功执行，Revit 保留外部命令所作的全部修改
Autodesk.Revit.UI.Result.Failed	外部命令未能完成任务，Revit 逆转外部命令所执行的操作。若已设置命令执行的消息参数，则 Revit 显示一个对话框文本 "Error - cannot be ignored "
Autodesk.Revit.UI.Result.Cancelled	用户取消了外部命令，Revit 逆转外部命令所作的更改。若已设置命令执行的消息参数，则 Revit 显示一个对话框文本 "Warning - can be ignored "

代码 1-12 显示了一条问候消息，并允许用户选择返回值。使用 Execute() 方法作为 Revit 应用程序入口。

**代码 1-12：给用户提示**

```
public Autodesk.Revit.UI.Result Execute (ExternalCommandData commandData,  
    ref string message, ElementSet elements)  
{  
    try  
    {  
        Document doc = commandData.Application.ActiveUIDocument.Document;  
        UIDocument uidoc = commandData.Application.ActiveUIDocument;  
        // Delete selected elements  
        ICollection<Autodesk.Revit.DB.ElementId> ids =
```



```

doc.Delete (uidoc.Selection.GetElementIds());

TaskDialog taskDialog = new TaskDialog ("Revit");
taskDialog.MainContent =
    ("Click Yes to return Succeeded. Selected members will be deleted.\n" +
    "Click No to return Failed. Selected members will not be deleted.\n" +
    "Click Cancel to return Cancelled. Selected members will not be deleted.");
TaskDialogCommonButtons buttons = TaskDialogCommonButtons.Yes |
    TaskDialogCommonButtons.No | TaskDialogCommonButtons.Cancel;
taskDialog.CommonButtons = buttons;
TaskDialogResult taskDialogResult = taskDialog.Show();

if (taskDialogResult == TaskDialogResult.Yes)
{
    return Autodesk.Revit.UI.Result.Succeeded;
}
else if (taskDialogResult == TaskDialogResult.No)
{
    elements = uidoc.Selection.Elements;
    message = "Failed to delete selection.";
    return Autodesk.Revit.UI.Result.Failed;
}
else
{
    return Autodesk.Revit.UI.Result.Cancelled;
}
}
catch
{
    message = "Unexpected Exception thrown.";
    return Autodesk.Revit.UI.Result.Failed;
}
}

```

(5) **IExternalCommandAvailability**。此接口可控制一个外部命令按钮是否可用。**IsCommandAvailable** 接口方法传递应用以及一组与 Revit 所选项类别相匹配的类别到应用。典型的用途是检查所选的类别，看它们是否满足命令运行条件。

在代码 1-13 中，当没有活动选项，或至少已选择了一面墙时，可用性检查允许按钮被点击。

#### 代码 1-13: Setting Command Availability

```

public class SampleAccessibilityCheck : IExternalCommandAvailability
{
    public bool IsCommandAvailable (AutodeskAutodesk.Revit.UI.UIApplication applicationData,
        CategorySet selectedCategories)
    {
        // Allow button click if there is no active selection
        if (selectedCategories.IsEmpty)

```





```
        return true;
        // Allow button click if there is at least one wall selected
        foreach (Category c in selectedCategories)
        {
            if (c.Id.IntegerValue == (int) BuiltInCategory.OST_Walls)
                return true;
        }
        return false;
    }
}
```

### 1.3.3 外部应用程序 (External Application)

开发人员可以通过外部应用程序和外部命令添加功能。功能区选项卡和功能区面板可使用外部应用程序进行定制，而功能区面板按钮则被绑定到外部命令。

要将外部应用程序添加到 Revit，需创建一个实现 `IExternalApplication` 接口的对象。

`IExternalApplication` 接口有两个在外部应用程序中重载的抽象方法，即 `OnStartup()` 和 `OnShutdown()`。Revit 在启动时调用 `OnStartup()`，关闭时调用 `OnShutdown()`。

代码 1-14 是 `OnStartup` 和 `OnShutdown` 的抽象概念定义。

#### 代码 1-14: `OnShutdown()` and `OnStartup()`

```
public interface IExternalApplication
{
    public Autodesk.Revit.UI.Result OnStartup (UIControlledApplication application);
    public Autodesk.Revit.UI.Result OnShutdown (UIControlledApplication application);
}
```

`UIControlledApplication` 参数提供对某些 Revit 事件的访问，允许定制功能区面板和控件，并添加功能区选项卡。例如，`UIControlledApplication` 的公共事件 `DialogBoxShowing` 可以用来捕获一个对话框显示事件。代码 1-15 注册了一个在显示对话框之前被调用的处理函数。

#### 代码 1-15: `DialogBoxShowing` Event

```
application.DialogBoxShowing += new
    EventHandler<Autodesk.Revit.Events.DialogBoxShowingEventArgs> (AppDialogShowing);
```

代码 1-16 阐释了如何使用 `UIControlledApplication` 类型注册一个事件处理程序，并在事件发生时进行处理。

#### 代码 1-16: 使用 `ControlledApplication`

```
public class Application_DialogBoxShowing : IExternalApplication
{
    // Implement the OnStartup method to register events when Revit starts.
    public Result OnStartup (UIControlledApplication application)
    {
        // Register related events
        application.DialogBoxShowing +=
            new EventHandler<Autodesk.Revit.UI.Events.DialogBoxShowingEventArgs> (AppDialogShowing);
    }
}
```



```

        return Result.Succeeded;
    }

    // Implement this method to unregister the subscribed events when Revit exits.
    public Result OnShutdown (UIControlledApplication application)
    {

        // unregister events
        application.DialogBoxShowing -=
        new EventHandler<Autodesk.Revit.UI.Events.DialogBoxShowingEventArgs> (AppDialogShowing);
        return Result.Succeeded;
    }

    // The DialogBoxShowing event handler, which allow you to
    // do some work before the dialog shows
    void AppDialogShowing (object sender, DialogBoxShowingEventArgs args)
    {

        // Get the help id of the showing dialog
        int dialogId = args.HelpId;

        // Format the prompt information string
        String promptInfo = "A Revit dialog will be opened.\n";
        promptInfo += "The help id of this dialog is " + dialogId.ToString() + "\n";
        promptInfo += "If you don't want the dialog to open, please press cancel button";

        // Show the prompt message, and allow the user to close the dialog directly.
        TaskDialog taskDialog = new TaskDialog ("Revit");
        taskDialog.MainContent = promptInfo;
        TaskDialogCommonButtons buttons = TaskDialogCommonButtons.Ok |
            TaskDialogCommonButtons.Cancel;
        taskDialog.CommonButtons = buttons;
        TaskDialogResult result = taskDialog.Show();
        if (TaskDialogResult.Cancel == result)
        {
            // Do not show the Revit dialog
            args.OverrideResult (1);
        }
        else
        {
            // Continue to show the Revit dialog
            args.OverrideResult (0);
        }
    }
}

```

#### 1.3.4 注册插件 ( Add-in Registration )

外部命令和外部应用程序需要注册才会出现在 Revit 中。通过将其添加到.addin 清单文件中即可完成注册。



外部命令和应用程序在 Revit 中的列表顺序取决于其在 Revit 启动时的读入顺序。

### 1. 清单文件 (Manifest Files)

自 Revit 2011 起, Revit API 提供了通过 .addin 清单文件注册 API 应用程序的功能。当清单文件被置于用户系统中以下两处之一时, Revit 会自动读取它们。

“应用程序数据”在非用户专有的位置:

- 对于 Windows XP 系统, 在 C: \Documents and Settings\All Users\Application Data\Autodesk\Revit\Addins\2014\。
- 对于 Vista/Windows 7 系统, 在 C: \ProgramData\Autodesk\Revit\Addins\2014\。

“应用程序数据”在用户专有的位置:

- 对于 Windows XP 系统, 在 C: \Documents and Settings\<user>\Application Data\Autodesk\Revit\Addins\2014\。
- 对于 Vista/Windows7 系统, 在 C: \Users\<user>\AppData\Roaming\Autodesk\Revit\Addins\2014\。

在这些位置上, Revit 在启动期间会读取并处理所有名为 .addin 的文件。用户专有或 all users 位置中的所有这些文件将一并考虑, 并按字母顺序加载。如果 all users 位置中的清单文件和用户专有位置中的清单文件重名, 则 all users 位置中的清单文件将被忽略。在每个清单文件内, 外部命令和外部应用程序会按它们的列表顺序加载。

添加一个外部命令的简单文件, 见代码 1-17。

#### 代码 1-17: Manifest .addin ExternalCommand

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<RevitAddIns>
  <AddIn Type="Command">
    <Assembly>c: \MyProgram\MyProgram.dll</Assembly>
    <AddInId>76eb700a-2c85-4888-a78d-31429ecae9ed</AddInId>
    <FullClassName>Revit.Samples.SampleCommand</FullClassName>
    <Text>Sample command</Text>
    <VendorId>ADSK</VendorId>
    <VendorDescription>Autodesk, www.autodesk.com</VendorDescription>
    <VisibilityMode>NotVisibleInFamily</VisibilityMode>
    <Discipline>Structure</Discipline>
    <Discipline>Architecture</Discipline>

    <AvailabilityClassName>Revit.Samples.SampleAccessibilityCheck</AvailabilityClassName>
    <LongDescription>
      <p>This is the long description for my command.</p>
      <p>This is another descriptive paragraph, with notes about how to use the command properly.</p>
    </LongDescription>
    <TooltipImage>c: \MyProgram\Autodesk.png</TooltipImage>
    <LargeImage>c: \MyProgram\MyProgramIcon.png</LargeImage>
  </AddIn>
</RevitAddIns>
```

添加一个外部应用程序的简单文件, 见代码 1-18。



代码 1-18: Manifest .addin ExternalApplication

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<RevitAddIns>
<AddIn Type="Application">
  <Name>SampleApplication</Name>
  <Assembly>c: \MyProgram\MyProgram.dll</Assembly>
  <AddInId>604B1052-F742-4951-8576-C261D1993107</AddInId>
  <FullClassName>Revit.Samples.SampleApplication</FullClassName>
  <VendorId>ADSK</VendorId>
  <VendorDescription>Autodesk, www.autodesk.com</VendorDescription>
</AddIn>
</RevitAddIns>
```

添加一个 DB 级外部应用程序的简单文件，见代码 1-19。

代码 1-19: Manifest .addin ExternalDBApplication

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<RevitAddIns>
<AddIn Type="DBApplication">
  <Assembly>c: \MyDBLevelApplication\MyDBLevelApplication.dll</Assembly>
  <AddInId>DA3D570A-1AB3-4a4b-B09F-8C15DFEC6BF0</AddInId>

  <FullClassName>MyCompany.MyDBLevelAddIn</FullClassName>

  <Name>My DB-Level AddIn</Name>
  <VendorId>ADSK</VendorId>
  <VendorDescription>Autodesk, www.autodesk.com</VendorDescription>
</AddIn>
</RevitAddIns>
```

单个清单文件里可以提供多个插件元素。

表 1-3 描述了可用的 XML 标记。

表 1-3 可用的 XML 标记

标记	说 明
Assembly	插件程序集文件的完整路径。所有外部命令和外部应用程序所必需
FullClassName	在程序集文件中的类的完整名，它实现外部命令接口或外部应用程序接口。所有外部命令和外部应用程序所必需
AddInId	GUID，表示此特定应用程序的 ID。对给定的 Revit 会话，AddInIds 必须是唯一的。Autodesk 建议为每个已注册的应用程序或命令生成一个唯一的 GUID。所有外部命令和外部应用程序所必需
Name	应用程序名，仅为外部应用程序所必需
Text	按钮的名称。仅外部命令使用此标记，且是可选的，默认值为"External Tool"
VendorId	符合 Autodesk 供应商 ID 标准的一串字符。所有外部命令和外部应用程序所必需。在 <a href="http://www.autodesk.com/symbreg">http://www.autodesk.com/symbreg</a> 向 Autodesk 注册供应商 ID 字符串
VendorDescription	描述包含供应商的法定名称和/或其他相关信息。可选项



续表

标记	说 明
Description	命令的简短描述, 将被用作按钮的工具提示。可选项, 只有外部命令使用这个标记。默认设置是只用命令文本作工具提示
VisibilityMode	外部命令在此模式可见。此选项有多个值可供设置。可选项, 仅外部命令用此标记。默认情况下显示所有模式中的命令, 包括在没有活动文件时。以前编写的需运行于活动文件的外部命令应作修改, 以确保代码能处理在没有活动文件时命令的调用, 或应用 <code>NotVisibleWhenNoActiveDocument</code> 模式。有关更多信息, 见表 1-4
Discipline	外部命令在此规程中可见, 此选项可设置多个值。可选项, 仅外部命令使用此标记。默认情况下显示所有规程中的命令。如果列出任何特定规程, 命令仅在这些规程中可见。有关更多信息, 见表 1-5
AvailabilityClassName	程序集文件中实现 <code>IEternalCommandAvailability</code> 接口的类的完整名称。该类允许根据上下文有选择地使命令按钮显示成灰色。可选项, 只有外部命令用此标记。默认设置是命令可用、始终可见
LargeImage	此图标用于外部工具下拉菜单按钮。可选项, 只有外部命令用此标记。默认设置是显示一个不带图标的按钮
SmallImage	当按钮提升到快速访问工具栏时使用此图标。可选项, 仅外部命令用此标记。默认设置是显示不带图标的快速访问工具栏按钮, 不带图标可能让用户难以理解
LongDescription	命令的详细说明, 将作为按钮的扩展工具提示的一部分, 当鼠标较长时间悬停于该命令时显示。可选项, 仅外部命令用此标记。若此属性和 <code>TooltipImage</code> 均未提供, 则该按钮将不会有扩展工具提示
TooltipImage	显示为按钮扩展工具提示的一部分的图像文件, 当鼠标较长时间悬停于该命令时会显示。可选项, 仅外部命令用此标记。若此属性和 <code>LongDescription</code> 均未提供, 则该按钮将不会有扩展工具提示
LanguageType	本地化设置外部工具按钮的 <code>Text</code> 、 <code>Description</code> 、 <code>LargeImage</code> 、 <code>LongDescription</code> 、 <code>TooltipImage</code> 。Revit 将从指定的语言资源 dll 加载资源值。该值可以是 Revit 所支持的 11 种语言之一。若未指定 <code>LanguageType</code> , 则会自动加载 Revit 当前会话正在使用的语言资源。有关详细信息, 请参阅 1.3.5 节本地化
AllowLoadIntoExistingSession	加载许可标志。设置为 <code>false</code> 以防止 Revit 在不重新启动的情况下从新添加的 .addin 清单文件中自动加载插件。可选项。默认设置是, Revit 在不重新启动的情况下从新添加的 .addin 清单文件自动加载插件

表 1-4

VisibilityMode 成员

成员名	说 明
AlwaysVisible	Revit API 支持的所有可能模式中, 命令都是可用的
NotVisibleInProject	有活动的项目文件时, 该命令不可见
NotVisibleInFamily	有活动的族文件时, 该命令不可见
NotVisibleWhenNoActiveDocument	无活动项目文件时, 该命令不可见

表 1-5

Discipline 成员

成员名称	说 明
Any	Revit API 支持的所有可能的规程中, 命令都是可用的
Architecture	Autodesk Revit Architecture 中该命令可见
Structure	Autodesk Revit Structure 中该命令可见
StructuralAnalysis	结构分析规程编辑工具可用时, 该命令可见



续表

成员名称	说 明
MassingAndSite	体量和场地规程编辑工具可用时, 该命令可见
EnergyAnalysis	能量分析规程编辑工具可用时, 该命令可见
Mechanical	机械规程编辑工具可用时, 该命令可见, 如在 Autodesk Revit MEP 中
Electrical	电气规程编辑工具可用时, 该命令可见, 如在 Autodesk Revit MEP 中
Piping	管道规程编辑工具可用时, 该命令可见, 如在 Autodesk Revit MEP 中
MechanicalAnalysis	机械分析规程编辑工具可用时, 该命令可见
PipingAnalysis	管道分析规程编辑工具可用时, 该命令可见
ElectricalAnalysis	电气分析规程编辑工具可用时, 该命令可见

2. .NET 插件清单文件实用程序 (.NET Add-in Utility for Manifest Files)

.NET 实用程序 DLL RevitAddInUtility.dll 提供了一个能读、写和修改 Revit 插件清单文件的专用 API, 见代码 1-20 和代码 1-21。它适合于从产品安装程序和脚本程序使用。请查阅 SDK 安装文件夹中的 RevitAddInUtility.chm 帮助文件中的 API 文档。

代码 1-20: 创建并编辑清单文件

```
//create a new addin manifest
RevitAddInManifest Manifest = new RevitAddInManifest( );

//create an external command
RevitAddInCommand command1 = new RevitAddInCommand ( "full path\\assemblyName.dll",
    Guid.NewGuid( ), "namespace.className");
command1.Description = "description";
command1.Text = "display text";

// this command only visible in Revit MEP, Structure, and only visible
// in Project document or when no document at all
command1.Discipline = Discipline.Mechanical | Discipline.Electrical |
    Discipline.Piping | Discipline.Structure;
command1.VisibilityMode = VisibilityMode.NotVisibleInFamily;

//create an external application
RevitAddInApplication application1 = new RevitAddInApplication ( "appName",
    "full path\\assemblyName.dll", Guid.NewGuid( ), "namespace.className");

//add both command (s) and application (s) into manifest
Manifest.AddInCommands.Add (command1);
Manifest.AddInApplications.Add (application1);

//save manifest to a file
RevitProduct revitProduct1 = RevitProductUtility.GetAllInstalledRevitProducts( ) [0];
Manifest.SaveAs (revitProduct1.AllUsersAddInFolder + "\\RevitAddInUtilitySample.addin");
```

**代码 1-21: 读取现有清单文件**

```
RevitProduct revitProduct1 = RevitProductUtility.GetAllInstalledRevitProducts() [0];

RevitAddInManifest revitAddInManifest =
    Autodesk.RevitAddIns.AddInManifestUtility.GetRevitAddInManifest (
        revitProduct1.AllUsersAddInFolder + "\\RevitAddInUtilitySample.addin");
```

**1.3.5 本地化 (Localization)**

Revit 可以将用户可见资源本地化为外部命令按钮, 包括文本、大图标图像、简单和详细说明, 以及工具提示图像。这需要创建一个 .NET 附属 DLL, 它包含按钮的字符串、图像和图标, 然后更改 .addin 文件中标记的值, 使其与附属 DLL 资源名称相对应, 但要前缀@字符。因此, 标记应由代码 1-22 变为代码 1-23。

**代码 1-22: 非本地化文本词条**

```
<Text>Extension Manager</Text>
```

**代码 1-23: 本地化文本词条**

```
<Text>@ExtensionText</Text>
```

这里, ExtensionText 是附属 DLL 中的资源名称。

附属 DLL 应在一个具有特定语言文化的语言名称目录中, 如 en 或 en-us。该目录必须位于包含插件程序集的目录中。请参阅 <http://msdn.microsoft.com/en-us/library/e9zazcx5.aspx> 创建托管附属 DLL。

通过 LanguageType 可以标记明确指定语言文字 (代码 1-24), 强制 Revit 使用特定语言资源 DLL, 而不管 Revit 会话采用何种语言。

**代码 1-24: 使用 LanguageType 标记**

```
<LanguageType>English_USA</LanguageType>
```

例如, 本地化外部命令清单文件中的成员时, 上述词条会强制 Revit 总是从 en-us 附属 dll 加载标记值, 而忽略当前 Revit 的语言文字设置。

Revit 支持在 Autodesk.Revit.ApplicationServices.LanguageType 定义的 11 种语言枚举类型: 美式英语、德语、西班牙语、法语、意大利语、荷兰语、简体中文、繁体中文、日语、韩语和俄语。

**1.3.6 属性 (Attributes)**

Revit API 提供了一些用于配置外部命令和外部应用程序行为的属性。

**1. 事务属性 (TransactionAttribute)**

自定义属性 Autodesk.Revit.Attributes.TransactionMode 必须应用于 IExternalCommand 接口的实现类以控制外部命令的事务行为。此选项没有默认值。当调用该命令时, 该方式控制 API 框架如何使用事务。受支持的值有:



- **TransactionModeAutomatic:** 外部命令执行前, Revit 将在活动文件中创建一个事务, 命令完成后事务将被提交或回滚 (基于 `ExternalCommand` 回调函数的返回值)。该命令无法创建和启动自身事务, 但可以创建子事务。该命令必须随结果返回值记录它的成功或失败状态。
- **TransactionModeManual:** Revit 不会创建事务 (但若外部命令返回一个 `Failure`, 则将创建一个外部事务组以回滚所有更改)。如果需要, 开发人员可以使用事务组合、子事务和事务组。必须遵循有关事务及其相关类的所有使用规则。事务名称还须给出, 它将会出现在撤销菜单中。Revit 会根据外部命令返回值检查所有事务 (以及事务组和子事务) 是否正常关闭。若否, Revit 将放弃对模型所作的任何更改。
- **TransactionModeReadOnly:** Revit 不会创建事务或事务组, 整个命令使用期内不可创建任何事务。外部命令只能使用模型的只读方法。如果命令试图启动事务 (或事务组), 或试图使用模型的写入方法, 则将引发异常。

`TransactionMode` 所有三种模式仅适用于活动文件。在命令进程中, 可以打开其他文件, 且可完全控制其事务、子事务和事务组的创建及使用 (即使处于只读模式)。

例如, 代码 1-25 设置了一个使用自动事务模式的外部命令。

**代码 1-25: TransactionAttribute**

```
[Transaction (TransactionModeAutomatic)]
public class Command : IExternalCommand
{
    public Autodesk.Revit.IExternalCommand.Result Execute (
        Autodesk.Revit.ExternalCommandData commandData,
        ref string message, Autodesk.Revit.DB.ElementSet elements)
    {
        // Command implementation, which modifies the active document directly
        // and no need to start/commit transaction.
    }
}
```

请参阅 5.2 节事务。

## 2. 日志属性 (JournalingAttribute)

自定义属性 `Autodesk.Revit.Attributes.JournalingAttribute` `JournalingAttribute` 可以有选择地应用于 `IExternalCommand` 接口的实现类, 以控制外部命令执行期间的日志记录行为。日志记录有两个选项:

- **JournalMode.NoCommandData:** `ExternalCommandData.JournalData` 映射的内容不会写入 Revit 日志。此选项允许 Revit API 按需要写入日志。此选项让用于选择或回应任务对话而调用 Revit 用户界面的命令能正确地重新执行。
- **JournalMode.UsingCommandData:** 使用命令数据中提供的 `IDictionaryString <String, String>`。这将隐藏外部命令调用和 `IDictionaryString <String, String>` 条目之间的所有 Revit 日志条目, 导致调用 Revit UI 进行选择或回应任务对话的命令不能正确地重复执行。如果未指定 `JournalingAttribute`, `JournalMode` 默认为 `UsingCommandData`,





见代码 1-26。

代码 1-26: JournalingAttribute

```
[Journaling (JournalingMode.UsingCommandData)]
public class Command : IExternalCommand
{
    public Autodesk.Revit.IExternalCommand.Result Execute (
        Autodesk.Revit.ExternalCommandData commandData,
        ref string message, Autodesk.Revit.DB.ElementSet elements)
    {
        return Autodesk.Revit.UI.Result.Succeeded;
    }
}
```

### 1.3.7 Revit 异常 (Revit Exceptions)

当 API 方法遇到非致命错误时会引发异常。异常应由 Revit 插件捕获。Revit API 帮助文件详细说明了特定方法常会遇到的异常。所有 Revit API 方法都会抛出一个 Autodesk.Revit.Exceptions.ApplicationException 的子类。这些异常与标准的 .NET 异常非常相像，如：

- 非法字符异常 (ArgumentException)。
- 非法操作异常 (InvalidOperationException)。
- 未找到文件异常 (FileNotFoundException)。

然而，尚有一些 Revit 独有的异常子类：

- 自动联结失败异常 (AutoJoinFailedException)。
- 重生成失败异常 (RegenerationFailedException)。
- 修改外部事务异常 (ModificationOutsideTransactionException)。

此外，还有一个称作 InternalException 的特殊异常类型，表示非预期的故障路径。这种类型的异常携带附加的诊断信息，可以传回到 Autodesk 以作诊断。

### 1.3.8 功能区面板和控件 (Ribbon Panels and Controls)

Revit 提供了集成自定义功能区面板和控件的 API 解决方案。IExternalApplication 可使用这些 API。自定义功能区面板可以添加到插件选项卡、分析选项卡或一个新的自定义功能区选项卡。

面板上可以包含按钮，大小均可，可以是简单的按钮，也可以是下拉式按钮或带有默认按钮的下拉分割按钮，下拉式按钮可执行多个命令。除以上这些按钮外，面板还可以包含单选按钮组、组合框和文本框。面板中可以使用垂直分隔符将命令按逻辑分组。最后，面板还可以包括一个滑出式控件，通过单击面板底部来访问该控件。

更多关于兼容 Autodesk 使用标准的用户界面开发信息，请参阅附录 E：API 用户界面指南章节中 E.9 节功能区指南。

#### 1. 新建功能区选项卡 (Create a New Ribbon Tab)

虽然功能区面板可以添加到插件或分析选项卡，但也可添加到一个新的自定义功能区选项卡，仅在必要时才使用此选项。为确保标准 Revit 功能区选项卡可见，自定义功能区



选项卡上限数量强制为 20 个。图 1-14 显示了有一个功能区面板和一些简单控件的新功能区选项卡。



图 1-14 新功能区选项卡

代码 1-27 是生成图 1-14 中功能区选项卡的代码。

#### 代码 1-27: 新建功能区选项卡

```
public Result OnStartup (UIControlledApplication application)
{
    // Create a custom ribbon tab
    String tabName = "This Tab Name";
    application.CreateRibbonTab (tabName);

    // Create two push buttons
    PushButtonData button1 = new PushButtonData ("Button1", "My Button #1",
        @"C:\ExternalCommands.dll", "Revit.Test.Command1");
    PushButtonData button2 = new PushButtonData ("Button2", "My Button #2",
        @"C:\ExternalCommands.dll", "Revit.Test.Command2");

    // Create a ribbon panel
    RibbonPanel m_projectPanel = application.CreateRibbonPanel (tabName, "This Panel Name");
    // Add the buttons to the panel
    List<RibbonItem> projectButtons = newList<RibbonItem>();
    projectButtons.AddRange (m_projectPanel.AddStackedItems (button1, button2));

    return Result.Succeeded;
}
```

## 2. 新建功能区面板和控件 (Create a New Ribbon Panel and Controls)

图 1-15 显示了应用各种功能区面板控件的插件选项卡上的功能区面板。以下各节将详细描述这些控件, 并提供创建功能区各个部分的代码示例。

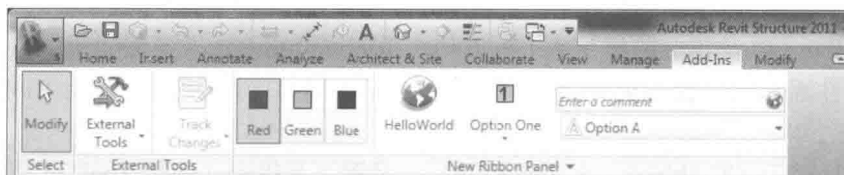


图 1-15 新功能区面板和控件

代码 1-28 概括了创建图 1-15 所示功能区面板所采取的步骤。本例中的每一个功能调用, 将随本节在稍后提供样例。这些样例假定程序集位于 D:\Sample\HelloWorld\bin\Debug\



Hello.dll, 包含以下外部命令类型:

- Hello.HelloButton。
- Hello.HelloOne。
- Hello.HelloTwo。
- Hello.HelloThree。
- Hello.HelloA。
- Hello.HelloB。
- Hello.HelloC。
- Hello.HelloRed。
- Hello.HelloBlue。
- Hello.HelloGreen。

代码 1-28: 功能区面板和控件

```
public Result OnStartup (Autodesk.Revit.UI.UIControlledApplication app)
{
    RibbonPanel panel = app.CreateRibbonPanel ("New Ribbon Panel");

    AddRadioGroup (panel);
    panel.AddSeparator ();
    AddPushButton (panel);
    AddSplitButton (panel);
    AddStackedButtons (panel);
    AddSlideOut (panel);

    return Result.Succeeded;
}
```

### 3. 功能区面板 (Ribbon Panel)

自定义功能区面板可以添加到插件选项卡 (默认) 或分析选项卡, 也可以添加到新的自定义功能区选项卡。有各种类型的功能区控件可以添加到功能区面板, 这将在下一节中详细讨论。所有功能区控件都有一些通用属性和功能。

(1) 功能区控件类 (Ribbon Control Classes)。每个功能区控件具有与之关联的两个类: 一个由 `RibbonItemData` (亦即 `SplitButtonData`) 派生, 用于创建控件并将其添加到功能区面板; 另一个则由 `RibbonItem` (亦即 `SplitButton`) 派生, 在它被添加到面板之后, 代表项目。 `RibbonItem` (以及相应的派生类) 可以使用 `RibbonItemData` (及派生类) 的可用属性。可以在该控件添加到面板之前设置这些属性, 或在其添加到面板之后使用 `RibbonItem` 类进行设置。

(2) 工具提示 (Tooltips)。大多数控件都可设置工具提示 (使用 `ToolTip` 属性), 当用户将鼠标移到该控件上方时会显示, 见图 1-16。当用户将鼠标悬停在控件上持续一段时间时, 将使用 `LongDescription` 和 `ToolTipImage` 属性来显示扩展工具提示。如果既未设置 `LongDescription`, 又未设置 `ToolTipImage`, 则不会显示扩展工具提示。如果未提供 `ToolTip`, 那么当鼠标移到该控件上方时, 会显示控件的文本 (`RibbonItem.ItemText`)。

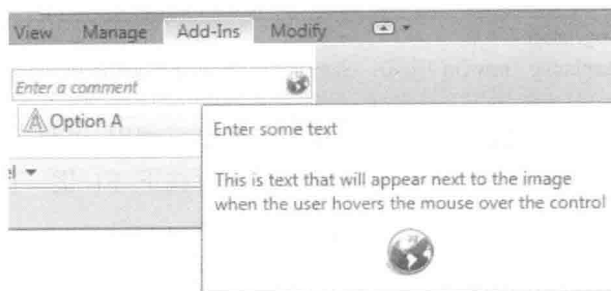


图 1-16 扩展工具提示

(3) 上下文相关帮助 (Contextual Help)。控件可以有与之关联的上下文帮助。当用户将鼠标悬停在控件上并按 F1 键时, 则会触发上下文帮助。上下文帮助选项包括链接到外部 URL、启动本地安装的帮助 (chm) 文件或链接到 Autodesk 帮助网站。ContextualHelp 类用于创建上下文帮助的类型, 然后 RibbonItem.SetContextualHelp() [或 RibbonItemData.SetContextualHelp()] 使其与控件关联。当 ContextualHelp 实例与某控件关联时, 若鼠标悬停在该控件上, 则工具提示的下方将出现文本“按 F1 键获得更多帮助”, 见图 1-17。

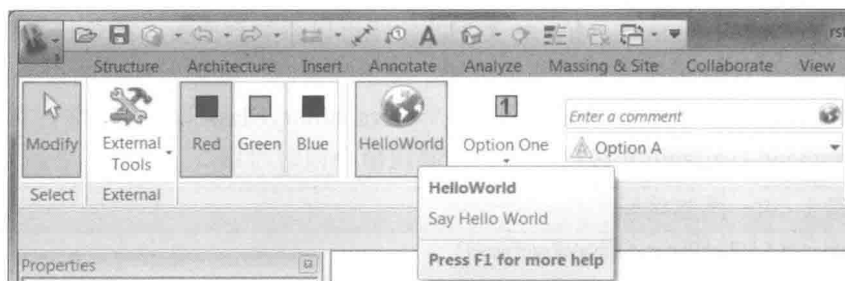


图 1-17 上下文帮助提示

代码 1-29 将一个新的上下文帮助与一个按钮控件相关联。当鼠标悬停在按钮上时按 F1 键, 将会在新浏览器窗口中打开 Autodesk 主页。

#### 代码 1-29: 上下文相关帮助

```
private void AddPushButton (RibbonPanel panel)
{
    PushButton pushButton = panel.AddItem (new PushButtonData ("HelloWorld",
        "HelloWorld", @"D:\Sample\HelloWorld\bin\Debug\HelloWorld.dll", "HelloWorld.CsHelloWorld")) as PushButton;

    // Set ToolTip and contextual help
    pushButton.ToolTip = "Say Hello World";
    ContextualHelp contextHelp = new ContextualHelp (ContextualHelpType.Url,
        "http://www.autodesk.com");
    pushButton.SetContextualHelp (contextHelp);

    // Set the large image shown on button
}
```



```
pushButton.LargeImage =  
    new BitmapImage (new Uri (@":D: \Sample\HelloWorld\bin\Debug\39-Globe_32x32.png"));
```

ContextualHelp 类有一个 Launch() 方法, 任何时候都可调用, 以显示由该 ContextualHelp 对象的内容所指定的帮助主题, 与控件处于活动状态时按下 F1 键一样。这可使帮助主题与插件程序创建的对话框内用户接口组件关联。

(4) 与控件关联的图像 (Associating Images with Controls)。使用 LargeImage 属性, 所有这些控件都可以有与之关联的图像。与大控件 (如非堆叠功能区和下拉按钮) 关联的图像, 最适合的尺寸是 32×32 像素, 而更大的图像将被调整以适应按钮尺寸。堆叠式按钮和小控件 (如文本框和组合框) 应该有一个 16×16 像素的图像集。大按钮也应有一个用于图像属性的 16×16 像素的图像集。如果要将命令移到快速访问工具栏, 则会使用该图像。如果未设置图像属性, 则命令移到快速访问工具栏时不会显示图像。请注意, 如果使用的图像大于 16×16 像素, 它不会被调整以适应工具栏。

(5) 功能区控件可用性 (Ribbon control availability)。功能区控件可以用 RibbonItem.Enabled 属性启用或禁用, 或用 RibbonItem.Visible 属性使其可见或不可见。

#### 4. 功能区控件 (Ribbon Controls)

除下列控件外, 垂直分隔符可以添加到功能区面板, 以将相关的控件分成组。

(1) 按钮 (Push Buttons)。有三种类型的按钮可以添加到面板: 简单按钮、下拉按钮和拆分按钮, 代码 1-30 示例了如何添加按钮。按下按钮时, 会触发相应的命令。

除 Enabled 属性, PushButton 还有一个 AvailabilityClassName 属性, 可用于设置 IExternal-CommandAvailability 接口名称, 控制该命令何时可用。

##### 代码 1-30: 添加按钮

```
private void AddPushButton (RibbonPanel panel)  
{  
    PushButton pushButton = panel.AddItem (new PushButtonData ("HelloWorld",  
        "HelloWorld", @":D: \HelloWorld.dll", "HelloWorld.CsHelloWorld")) as PushButton;  
  
    pushButton.ToolTip = "Say Hello World";  
    // Set the large image shown on button  
    pushButton.LargeImage =  
        new BitmapImage (new Uri (@":D: \Sample\HelloWorld\bin\Debug\39-Globe_32x32.png"));  
}
```

(2) 下拉按钮 (Drop-down Buttons)。下拉按钮可展开显示下拉菜单中两个或更多命令。Revit API 中下拉按钮以 PulldownButtons 表示。下拉菜单项目之间, 可添加水平分隔符。

下拉菜单中的每个命令也都可以有一个与之关联的大图像, 如代码 1-30 所示。

(3) 拆分按钮 (Split Buttons)。拆分按钮是个有默认附加按钮的下拉按钮, 代码 1-31 示例了如何添加拆分按钮。按钮的上半部分作用如同简单按钮, 而下半部分起到下拉按钮的作用。最初, 按钮是下拉列表中的第一项。然而, 通过使用 IsSynchronizedWithCurrentItem 属性, 默认命令 (显示为拆分按钮的上半部分) 将同步为上次使用的命令。默认情况下它也会同步。从图 1-15 拆分按钮中选择 Option Two 选项将产生图 1-18 所示结果。

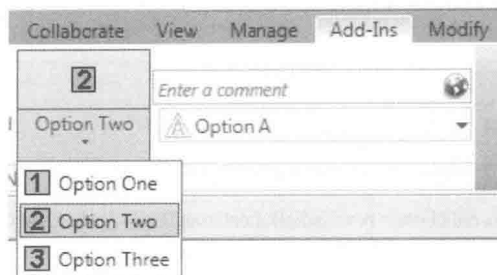


图 1-18 分隔按钮与当前项同步

注意：对于拆分按钮，ToolTip、ToolTipImage 和 LongDescription 属性将被忽略，而代之以显示当前按钮的工具提示。

#### 代码 1-31：添加拆分按钮

```
private void AddSplitButton (RibbonPanel panel)
{
    string assembly = @"D: \Sample\HelloWorld\bin\Debug\Hello.dll";

    // create push buttons for split button drop down
    PushButtonData bOne = new PushButtonData ("ButtonNameA", "Option One",
        assembly, "Hello.HelloOne");
    bOne.LargeImage =
        new BitmapImage (new Uri (@@"D: \Sample\HelloWorld\bin\Debug\One.bmp"));

    PushButtonData bTwo = new PushButtonData ("ButtonNameB", "Option Two",
        assembly, "Hello.HelloTwo");
    bTwo.LargeImage =
        new BitmapImage (new Uri (@@"D: \Sample\HelloWorld\bin\Debug\Two.bmp"));

    PushButtonData bThree = new PushButtonData ("ButtonNameC", "Option Three",
        assembly, "Hello.HelloThree");
    bThree.LargeImage =
        new BitmapImage (new Uri (@@"D: \Sample\HelloWorld\bin\Debug\Three.bmp"));

    SplitButtonData sb1 = new SplitButtonData ("splitButton1", "Split");
    SplitButton sb = panel.AddItem (sb1) as SplitButton;
    sb.AddPushButton (bOne);
    sb.AddPushButton (bTwo);
    sb.AddPushButton (bThree);
}
```

(4) 单选按钮 (Radio Buttons)。单选按钮组是一组互斥的开关按钮，一次只可选定一个。在 `RadioButtonGroup` 添加到面板之后，使用 `AddItem()` 或 `AddItems()` 方法将开关按钮添加到组内。开关按钮由 `PushButton` 派生，`RadioButtonGroup.Current` 属性可用于访问当前所选按钮。代码 1-32 示例了如何添加单选按钮组。

注意：工具提示不适用于单选按钮组。当鼠标移动到某按钮上方时，代之以显示该开



关按钮的工具提示。

**代码 1-32: 添加单选按钮组**

```
private void AddRadioGroup (RibbonPanel panel)
{
    // add radio button group
    RadioButtonGroupData radioData = new RadioButtonGroupData ("radioGroup");
    RadioButtonGroup radioButtonGroup = panel.AddItem (radioData) as RadioButtonGroup;

    // create toggle buttons and add to radio button group
    ToggleButtonData tb1 = new ToggleButtonData ("toggleButton1", "Red");
    tb1.ToolTip = "Red Option";
    tb1.LargeImage = new BitmapImage (new Uri (@":D: \Sample\HelloWorld\bin\Debug\Red.bmp"));
    ToggleButtonData tb2 = new ToggleButtonData ("toggleButton2", "Green");
    tb2.ToolTip = "Green Option";
    tb2.LargeImage = new BitmapImage (new Uri (@":D: \Sample\HelloWorld\bin\Debug\Green.bmp"));
    ToggleButtonData tb3 = new ToggleButtonData ("toggleButton3", "Blue");
    tb3.ToolTip = "Blue Option";
    tb3.LargeImage = new BitmapImage (new Uri (@":D: \Sample\HelloWorld\bin\Debug\Blue.bmp"));
    radioButtonGroup.AddItem (tb1);
    radioButtonGroup.AddItem (tb2);
    radioButtonGroup.AddItem (tb3);
}
```

(5) 文本框 (Text Box)。文本框是用户输入文本的输入控件。通过将 `ShowImageAsButton` 属性设置为 `true`, 文本框图像可以作为一个可点击的按钮。默认属性为 `false`。当属性为 `false` 时, 图像显示在文本框的左侧, 当它作为一个可点击的按钮, 会显示在文本框的右端。

只有当用户按下 `Enter` 键, 或在图像显示为按钮时单击关联的图像时, 文本框中输入的文本才被接受。否则, 文本将恢复为它先前的值。

除提供文本框工具提示外, `PromptText` 属性可用于指示用户在文本框中输入何种类型的信息。当文本框为空并且无键盘焦点事件时, 会显示提示文本。提示文本以斜体显示。在图 1.15 的文本框中提示文本为 "Enter a comment"。

文本框的宽度可以使用 `Width` 属性设置。默认值是 200 个与设备无关的单位。

当用户按下 `Enter` 键, 或当 `ShowImageAsButton` 设置为 `true` 时, 点击与文本框相关联的图像, 会触发 `TextBox.EnterPressed` 事件。在执行 `EnterPressed` 事件处理程序时, `sender` 对象强制转换为 `TextBox` 以获取用户输入的值, 见代码 1-33。

**代码 1-33: TextBox.EnterPressed 事件处理程序**

```
void ProcessText (object sender, Autodesk.Revit.UI.Events.TextBoxEnterPressedEventArgs args)
{
    // cast sender as TextBox to retrieve text value
    TextBox textBox = sender as TextBox;
    string strText = textBox.Value as string;
}
```

继承的 `ItemText` 属性对文本框没有影响。用户输入的文本可以从 `Value` 属性获取, 但



必须将它转换为字符串。

作为向功能区面板添加文本框的例子，参见下文堆叠面板项目章节，其中还包括了如何注册事件。

(6) 组合框 (Combo Box)。组合框是带有一组可选项的下拉列表。在组合框添加到面板之后，使用 `AddItem()` 或 `AddItems()` 方法添加组合框成员到列表。也可以添加分隔符来分隔列表中的项目，或使用 `ComboBox.MemberGroupName` 属性对成员有选择性地分组。具有相同组名的所有成员将分在一组中，并有一个显示组名的表头，见图 1-19。未指定组名的任何项目将放在列表的顶部。请注意，对项目进行分组时，不应使用分隔符，因为它们将被置于分组的末尾，而不是按它们被添加的顺序放置。

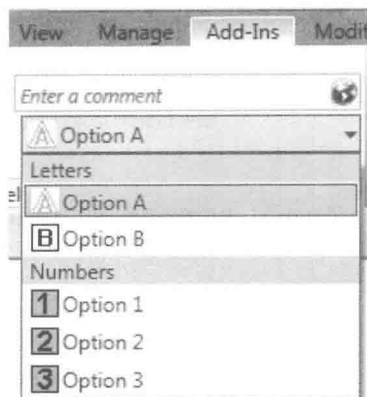


图 1-19 带分组的组合框

组合框有三个事件：

- **CurrentChanged**: 在组合框的当前项被改变时触发。
- **DropDownClosed**: 在组合框的下拉列表被关闭时触发。
- **DropDownOpened**: 在组合框的下拉列表被打开时触发。

参阅下文有关堆叠面板项目的代码，作为向功能区面板添加组合框的例子。

(7) 堆叠面板项目 (Stacked Panel Items)。为了节省面板空间，可以在两个或三个堆叠中添加小面板项目。堆叠中的每个项目可以是按钮、下拉按钮、组合框或文本框。单选按钮组和拆分按钮不能堆叠。堆叠按钮应该有与其关联的图像，通过 `Image` 属性获得，而不是 `LargeImage`。16×16 图像是理想的堆叠小按钮。

代码 1-34 是生成堆叠文本框和组合框的代码。

#### 代码 1-34: Adding a text box and combo box as stacked items

```
private void AddStackedButtons (RibbonPanel panel)
{
    ComboBoxData cbData = new ComboBoxData ("comboBox");

    TextBoxData textData = new TextBoxData ("Text Box");
    textData.Image =
        new BitmapImage (new Uri (@":D: \Sample\HelloWorld\bin\Debug\39-Globe_16x16.png"));
    textData.Name = "Text Box";
    textData.ToolTip = "Enter some text here";
    textData.LongDescription = "<p>This is text that will appear next to the image</p>"
        + "<p>when the user hovers the mouse over the control</p>";
    textData.ToolTipImage =
        new BitmapImage (new Uri (@":D: \Sample\HelloWorld\bin\Debug\39-Globe_32x32.png"));

    IList<RibbonItem> stackedItems = panel.AddStackedItems (textData, cbData);
    if (stackedItems.Count > 1)
    {
        TextBox tBox = stackedItems [0] as TextBox;
        if (tBox != null)
```





```
{
    tBox.PromptText = "Enter a comment";
    tBox.ShowImageAsButton = true;
    tBox.ToolTip = "Enter some text";
    // Register event handler ProcessText
    tBox.EnterPressed +=
new EventHandler<Autodesk.Revit.UI.Events.TextBoxEnterPressedEventArgs> (ProcessText);
}

ComboBox cBox = stackedItems [1] as ComboBox;
if (cBox != null)
{
    cBox.ItemText = "ComboBox";
    cBox.ToolTip = "Select an Option";
    cBox.LongDescription = "Select a number or letter";

    ComboBoxMemberData cboxMemDataA = new ComboBoxMemberData ("A", "Option A");
    cboxMemDataA.Image =
        new BitmapImage (new Uri (@":D: \Sample\HelloWorld\bin\Debug\A.bmp"));
    cboxMemDataA.GroupName = "Letters";
    cBox.AddItem (cboxMemDataA);

    ComboBoxMemberData cboxMemDataB = new ComboBoxMemberData ("B", "Option B");
    cboxMemDataB.Image =
        new BitmapImage (new Uri (@":D: \Sample\HelloWorld\bin\Debug\B.bmp"));
    cboxMemDataB.GroupName = "Letters";
    cBox.AddItem (cboxMemDataB);

    ComboBoxMemberData cboxMemData = new ComboBoxMemberData ("One", "Option 1");
    cboxMemData.Image =
        new BitmapImage (new Uri (@":D: \Sample\HelloWorld\bin\Debug\One.bmp"));
    cboxMemData.GroupName = "Numbers";
    cBox.AddItem (cboxMemData);
    ComboBoxMemberData cboxMemData2 = new ComboBoxMemberData ("Two", "Option 2");
    cboxMemData2.Image =
        new BitmapImage (new Uri (@":D: \Sample\HelloWorld\bin\Debug\Two.bmp"));
    cboxMemData2.GroupName = "Numbers";
    cBox.AddItem (cboxMemData2);
    ComboBoxMemberData cboxMemData3 = new ComboBoxMemberData ("Three", "Option 3");
    cboxMemData3.Image =
        new BitmapImage (new Uri (@":D: \Sample\HelloWorld\bin\Debug\Three.bmp"));
    cboxMemData3.GroupName = "Numbers";
    cBox.AddItem (cboxMemData3);
}
}
```

(8) 滑出式面板 (Slide-out Panel)。使用 `RibbonPanel.AddSlideOut()` 方法将滑出式面板添加到功能区面板的底部。添加滑出式面板后, 在面板的底部显示一个箭头, 点击它时会显



示滑出选项。调用 `AddSlideOut()` 后, 将新项目添加到面板的后续调用会把项目添加到该滑出式面板, 所以滑出式面板必须在所有其他控件已添加到功能区面板之后再添加, 见图 1-20。



图 1-20 滑出式面板

代码 1-35 将产生图 1-20 所示的滑出式面板。

#### 代码 1-35: `TextBox.EnterPressed` event handler

```
private static void AddSlideOut (RibbonPanel panel)
{
    string assembly = @"D: \Sample\HelloWorld\bin\Debug\Hello.dll";

    panel.AddSlideOut ();

    // create some controls for the slide out
    PushButtonData b1 = new PushButtonData ("ButtonName1", "Button 1",
        assembly, "Hello.HelloButton");
    b1.LargeImage =
        new BitmapImage (new Uri (@@"D: \Sample\HelloWorld\bin\Debug\39-Globe_32x32.png"));
    PushButtonData b2 = new PushButtonData ("ButtonName2", "Button 2",
        assembly, "Hello.HelloTwo");
    b2.LargeImage =
        new BitmapImage (new Uri (@@"D: \Sample\HelloWorld\bin\Debug\39-Globe_16x16.png"));

    // items added after call to AddSlideOut() are added to slide-out automatically
    panel.AddItem (b1);
    panel.AddItem (b2);
}
```

### 1.3.9 Revit 式任务对话框 (Revit-style Task Dialogs)

Revit 式任务对话框可替代简单 Windows 消息框 (图 1-21), 它可以用于显示信息和接收用户的简单输入。它有一组常见的控件, 按标准顺序排列以确保与 Revit 其他部分的外观相协调。

有两种方法来创建和显示任务对话框。第一种方法是构建 `TaskDialog`, 单独设置其属性, 并使用实例方法 `Show()` 显示给用户; 第二种方法是使用静态 `Show()` 中的一个方法,



一步构建和显示对话框。使用静态方法只可指定一个选项子集。

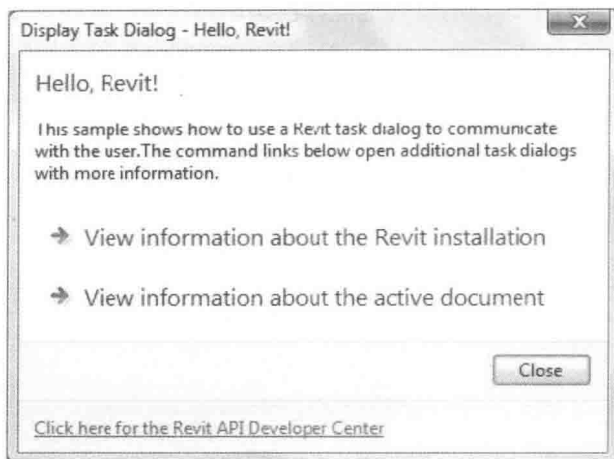


图 1-21 Revit 式对话框

更多关于开发符合 Autodesk 使用标准的任务对话框信息，请参见附录 E：API 用户界面指南章节中的任务对话框。

代码 1-36 演示了如何创建和显示图 1-21 所示的任务对话框。

#### 代码 1-36: Displaying Revit-style TaskDialog

```
[Autodesk.Revit.Attributes.Transaction (Autodesk.Revit.Attributes.TransactionModeAutomatic)]
class TaskDialogExample : IExternalCommand
{
    public Autodesk.Revit.UI.Result Execute (ExternalCommandData commandData, ref string message, Autodesk.Revit.DB.ElementSet elements)
    {
        // Get the application and document from external command data.
        Application app = commandData.Application.Application;
        Document activeDoc = commandData.Application.ActiveUIDocument.Document;

        // Creates a Revit task dialog to communicate information to the user.
        TaskDialog mainDialog = new TaskDialog ("Hello, Revit!");
        mainDialog.MainInstruction = "Hello, Revit!";
        mainDialog.MainContent =
            "This sample shows how to use a Revit task dialog to communicate with the user."
            + "The command links below open additional task dialogs with more information.";

        // Add commandLink options to task dialog
        mainDialog.AddCommandLink (TaskDialogCommandLinkId.CommandLink1,
            "View information about the Revit installation");
        mainDialog.AddCommandLink (TaskDialogCommandLinkId.CommandLink2,
            "View information about the active document");

        // Set common buttons and default button. If no CommonButton or CommandLink is added,
        // task dialog will show a Close button by default
    }
}
```



```

mainDialog.CommonButtons = TaskDialogCommonButtons.Close;
mainDialog.DefaultButton = TaskDialogResult.Close;

// Set footer text. Footer text is usually used to link to the help document.
mainDialog.FooterText =
    "" + "Click here for the Revit API Developer Center";

TaskDialogResult tResult = mainDialog.Show();

// If the user clicks the first command link, a simple Task Dialog
// with only a Close button shows information about the Revit installation.
if (TaskDialogResult.CommandLink1 == tResult)
{
    TaskDialog dialog_CommandLink1 = new TaskDialog ("Revit Build Information");
    dialog_CommandLink1.MainInstruction =
        "Revit Version Name is: " + app.VersionName + "\n"
        + "Revit Version Number is: " + app.VersionNumber + "\n"
        + "Revit Version Build is: " + app.VersionBuild;

    dialog_CommandLink1.Show();
}

// If the user clicks the second command link, a simple Task Dialog
// created by static method shows information about the active document
else if (TaskDialogResult.CommandLink2 == tResult)
{
    TaskDialog.Show ("Active Document Information",
        "Active document: " + activeDoc.Title + "\n"
        + "Active view name: " + activeDoc.ActiveView.Name);
}

return Autodesk.Revit.UI.Result.Succeeded;
}
}

```

### 1.3.10 数据库级外部应用程序 (DB-level External Applications)

数据库级插件是不向 Revit 用户界面添加任何内容的外部应用程序。当应用程序的目的是为 Revit 会话分配事件和/或更新时, 可以使用数据库级外部应用程序。

为了将数据库级外部应用程序添加到 Revit, 需要创建一个实现 `IEternalDBApplication` 接口的对象。

`IEternalDBApplication` 接口有两个抽象方法, 即 `OnStartup()` 和 `OnShutdown()`, 在数据库级外部应用程序中重载, 见代码 1-37。当 Revit 启动时调用 `OnStartup()`, 关闭时调用 `OnShutdown()`。这与外部应用程序非常相似, 但注意这些方法返回 `Autodesk.Revit.DB.ExternalDBApplicationResult` 而不是 `Autodesk.Revit.UI.Result`, 使用 `ControlledApplication`



而不是 `UIControlledApplication`。

代码 1-37: 外部应用程序 `OnShutdown()` 和 `OnStartup()`

```
public interface IExternalDBApplication
{
    public Autodesk.Revit.DB.ExternalDBApplicationResult OnStartup (ControlledApplication application);
    public Autodesk.Revit.DB.ExternalDBApplicationResult OnShutdown (ControlledApplication application);
}
```

`ControlledApplication` 参数提供对 Revit 数据库事件的访问。数据库级应用程序响应的事件和更新, 可在 `OnStartup()` 方法中注册。

## 1.4 应用程序和文件 (Application and Document)

应用程序和文件是 Revit 平台 API 的顶层对象。`Application`、`UIApplication`、`Document` 和 `UIDocument` 类代表了这些对象。

- 应用程序对象是指一个独立的 Revit 会话, 提供访问文件、选项和应用程序范围的其他数据及设置。
  - `Autodesk.Revit.UI.UIApplication` 为应用程序提供用户界面级的界面访问, 包括添加功能区面板到用户界面, 并能在用户界面中获取活动文件。
  - `Autodesk.Revit.ApplicationServices.Application` 提供对应用程序级的所有其他属性的访问。
- 文件对象是表示建筑模型的单个 Revit 项目文件。Revit 可以有多个打开的项目和一个项目的多个视图。
  - `Autodesk.Revit.UI.UIDocument` 为文件提供用户界面级的界面访问, 如选择内容时提示用户作出选择及选取点。
  - `Autodesk.Revit.DB.Document` 提供对文件级的所有其他属性的访问。
- 如果打开了多个文件, 则活动文件的视图在 Revit 会话中处于活动状态。

本节分述所有应用程序和文件的功能, 重点介绍文件管理、设置和单位。有关图元类的详细信息, 请参阅 1.5 节图元概要和 2.5 节编辑图元; 有关视图图元的详细信息, 请参阅 2.6 节视图。

### 1.4.1 应用程序功能 (Application Functions)

应用程序功能提供对文件、对象和其他应用程序数据及设置的访问。以下各节说明和定义了所有应用程序和用户界面应用程序功能。

#### 1. 应用程序 (Application)

该类表示 Autodesk Revit 应用程序, 提供对文件、选项和应用程序范围的其他数据和设置的访问。

(1) 应用程序版本信息 (Application Version Information)。应用程序属性包括 `VersionBuild`、`VersionNumber` 和 `VersionName`。这些属性可用于提供基于 Revit 构建版本的插件特性, 请参阅本节后续内容: 如何使用应用程序属性强制插件使用合适版本。



(2) 应用程序全局设置 (Application-wide Settings)。SharedParametersFilename 属性、GetLibraryPaths() 和 SetLibraryPaths() 方法提供对这些应用程序全局设置的访问。

(3) 文件管理 (Document Management)。应用程序类提供了创建以下类型文件的方法：

- 族文件。
- 项目文件。
- 项目样板。

OpenDocumentFile() 方法可以用来打开任何这些类型的文件。使用 Documents 属性可以检索所有打开的文件。有关详细信息，请参阅 1.4.3 节文档及文件管理。

(4) 会话信息 (Session Information)。属性 (如 UserName) 和方法 [如 GetRevitServerNetworkHosts()]，提供了特定会话信息的只读访问。

(5) 共享参数管理 (Shared Parameter Management)。Revit 每次使用一个共享参数文件。Application.OpenSharedParameterFile() 方法访问共享参数文件，其路径在 SharedParametersFilename 属性中设置。有关详细信息，请参阅 5.1.1 节共享参数。

(6) 事件 (Events)。应用程序类公开了文件和应用程序事件，如文件打开和保存。当事件已启用时，订阅这些事件通知应用程序并据此执行。有关详细信息，请参见 1.3 节插件集成部分中的访问 Revit 事件。

(7) 创建 (Create)。Create 属性返回一个 Object Factory，用于在 Revit 平台 API 中创建应用程序范围的实用程序和几何对象。创建一个 Revit 应用内存中的对象，而不是应用程序内存中的对象时，使用创建属性。

(8) 故障发布及处理 (Failure Posting and Handling)。可从静态方法 GetFailureDefinitionRegistry() 获取 FailureDefinitionRegistry，它包含所有已注册的 FailureDefinitions。而静态方法 RegisterFailuresProcessor() 可用于注册自定义的 IFailuresProcessor。更多信息，请参阅 5.8 节进阶专题中的故障发布和处理。

## 2. 用户界面应用程序 (UIApplication)

此类表示一个 Autodesk Revit 用户界面的活动会话，提供对用户界面定制方法、事件和活动文件的访问。

(1) 文件管理 (Document Management)。UIApplication 使用 UIActiveDocument 属性提供对活动文件的访问。此外，可以使用重载 OpenAndActivateDocument() 方法打开 Revit 文件。文件将默认以活动视图打开。此方法不能在事务内部调用，只能在事件期间调用，而且 Revit 不能有已打开的其他活动文件，且该事件不能是另一个事件的嵌套事件。

(2) 插件管理 (Add-in Management)。ActiveAddInId 属性获取当前活动的外部应用程序或外部命令 ID，而 LoadedApplications 属性返回一组已成功载入的外部应用程序。

(3) 功能区面板实用程序 (Ribbon Panel Utility)。使用 UIApplication 对象向 Revit 添加新的功能区面板和控件。有关详细信息，请参见 1.3 节插件集成中的功能区面板和控件部分。

(4) 范围 (Extents)。DrawingAreaExtents 属性返回一个以屏幕像素坐标表示的矩形绘图区域，而 MainWindowExtents 属性返回一个以屏幕像素坐标表示的矩形 Revit 主窗口。

(5) 事件 (Events)。UIApplication 类公开了与用户界面相关的事件。例如，显示一个对话框。当事件已启用时，订阅这些事件通知应用程序并据此执行。有关详细信息，请参



见 1.3 节插件集成部分中的访问 Revit 事件。

### 3. 规程控件 (Discipline Controls)

属性:

- Application.IsArchitectureEnabled。
- Application.IsStructureEnabled。
- Application.IsStructuralAnalysisEnabled。
- Application.IsMassingEnabled。
- Application.IsEnergyAnalysisEnabled。
- Application.IsSystemsEnabled。
- Application.IsMechanicalEnabled。
- Application.IsMechanicalAnalysisEnabled。
- Application.IsElectricalEnabled。
- Application.IsElectricalAnalysisEnabled。
- Application.IsPipingEnabled。
- Application.IsPipingAnalysisEnabled。

以上属性提供读取和修改相关规程方面的访问权限。应用程序可以读取这些属性以确定何时启用或禁用其用户界面方面的功能。

当切换规程状态时, 会相应调整 Revit 用户界面, 并酌情启用或禁用某些操作和功能。启用分析模式仅当相应规程已启用时才会生效。例如, 除非机械规程也已启用, 否则启用机械分析将不会生效。

### 4. 如何使用应用程序属性强制插件使用合适版本

由于存在特定的修补程序或兼容 API, 有时需要插件仅运行于某个特定的 Revit 更新版本。代码 1-38 演示了一个技术手法, 用来确定 Revit 是否是已知的 Revit 初始版本之后的某个更新版本。

#### 代码 1-38: 使用版本构建来识别插件是否兼容

```
public void GetVersionInfo (Autodesk.Revit.ApplicationServices.Application app)
{
    // 20110309_2315 is the datecode of the initial release of Revit 2012
    if (app.VersionNumber == "2012" &&
        String.Compare (app.VersionBuild, "20110309_2315") > 0)
    {
        TaskDialog.Show ("Supported version",
            "This application supported in this version.");
    }
    else
    {
        TaskDialog dialog = new TaskDialog ("Unsupported version.");
        dialog.MainIcon = TaskDialogIcon.TaskDialogIconWarning;
        dialog.MainInstruction = "This application is only supported in Revit 2012 UR1 and later.";
        dialog.Show();
    }
}
```



### 1.4.2 文件功能 (Document Functions)

Document 存储 Revit 图元, 管理数据并更新多个数据视图。Document 类主要提供以下功能。

#### 1. 文件 (Document)

Document 类表示打开的 Autodesk Revit 项目。

(1) 设置属性 (Settings Property)。Settings 属性返回一个对象, 该对象提供对 Revit 项目内全体构件的访问。有关详细信息, 请参阅 1.4.4 节设置。

(2) 地点和位置 (Place and Locations)。每个项目只有一个场地位置, 标识项目在地球上的物理定位。一个项目可以有多个项目位置, 每个位置都是其场地位置的偏移或旋转。有关详细信息, 请参阅进阶专题中 5.12 节地点和位置。

(3) 类型集合 (Type Collections)。Document 提供属性, 如 FloorTypes、WallTypes 等。所有属性都会返回一个包含载入该项目的相关类型的对象集合。

(4) 视图管理 (View Management)。项目文件可以有多个视图。ActiveView 属性返回一个代表活动视图的视图对象。通过过滤项目中的图元可以检索其他视图。有关详细信息, 请参阅 2.6 节视图。

(5) 图元检索 (Element Retrieval)。Document 对象储存项目中的图元。可用 Element 属性的 ElementId 或 UniqueId 检索特定图元。有关详细信息, 请参见 1.5 节图元概要。

(6) 文件管理 (File Management)。每个 Document 对象代表一个 Revit 项目文件。Document 提供以下功能:

- 检索文件信息, 如文件路径名称和项目名称。
- 提供 Close() 和 Save() 方法来关闭和保存文件。

有关详细信息, 请参阅 1.4.3 节文档及文件管理。

(7) 图元管理 (Element Management)。Revit 维护项目中的所有图元对象。要创建新图元, 请使用 Create 属性, 它返回一个 Object Factory, 用于在 Revit 平台 API 中创建项目的新图元实例, 如 FamilyInstance 或 Group。

Document 类也可用于删除图元。要删除项目中的某个图元, 请使用 Delete() 方法。被删除的图元及其任何从属图元将不再显示, 并从文件中删除。引用已删除的图元是无效的, 且会引发异常。有关详细信息, 请参阅 2.5 节编辑图元。

(8) 事件 (Events)。事件由某些操作引发, 例如, 使用 Save 或 Save As 保存项目。要捕获应用程序中的事件并作出响应, 必须注册事件处理程序。有关详细信息, 请参阅进阶专题中 5.3 节事件。

(9) 文件状态 (Document Status)。一些提供有关文件状态信息的属性如下:

- IsModifiable: 文件可否修改。
- IsModified: 自从打开或保存后文件是否已被更改。
- IsReadOnly: 当前文件是只读还是可修改。
- IsReadOnlyFile: 文件是否是以只读方式打开的。
- IsFamilyDocument: 文件是否是族文件。
- IsWorkshared: 是否已启用文件工作集。





(10) 其他 (Others)。Document 还提供了一些其他功能：

- **ParameterBindings** 属性：参数定义和类别之间的映射。有关详细信息，请参阅 5.1.1 节共享参数。
- **ReactionsAreUpToDate** 属性：报告反作用荷载是否已改变。有关详细信息，请参阅 Revit Structure 章节中 4.2.3 节荷载。

2. 用户界面文件 (UIDocument)

UIDocument 类表示在 Revit 用户界面中打开的一个 Autodesk Revit 项目。

(1) 图元检索 (Element Retrieval)。UIDocument 中使用 **Selection** 属性检索所选图元。此属性返回一个对象，该对象表示包含所选项目图元的活动选集。它还提供了在 Revit 模型中选择对象的用户界面交互方法。有关详细信息，请参见 1.5 节图元概要。

(2) 图元显示 (Element Display)。ShowElements() 方法使用缩放以便显示关注的图元。

(3) 视图管理 (View Management)。通过调用 RefreshActiveView() 方法，UIDocument 类可用于刷新活动文件中的活动视图。ActiveView 属性可用于检索或设置该文件的活动视图。更改当前视图有一些限制，只能用于活动文件，且不能是只读状态，不能在事务中。此外，以下事件期间的活动视图也不能更改：ViewActivatin、ViewActivated 事件，或任何操作前事件，如 DocumentSaving、DocumentClosing，或其他类似事件。

UIDocument 类还可以用来获取 Revit 用户界面中所有已打开视图窗口的列表。GetOpenUIViews() 方法返回一个包含 Revit 用户界面中视图窗口有关数据的 UIViews 列表。

1.4.3 文档及文件管理 (Document and File Management)

文档及文件管理使开发人员可轻松地创建并找到文件。

1. 文件检索 (Document Retrieval)

Application 类维护所有文件。如前所述，可以在一个会话中打开多个文件。检索活动文件使用 UIApplication 类的 ActiveUIDocument 属性。

所有已打开的文件，包括活动文件，使用 Application 类的 Documents 属性来检索。该属性返回一个包含 Revit 会话中已打开的所有文件集合。

2. 文档文件信息 (Document File Information)

Document 类为每个文件提供了两个相应属性：PathName 和 Title。

- **PathName** 返回文件的完整路径。若项目自创建后从未保存过，则 PathName 返回一个空字符串。
- **Title** 是项目的名称，通常来自项目文件名。返回值因系统设置而有所不同。

3. 打开文件 (Open a Document)

Application 类提供了一个重载方法来打开现有项目文件，该方法及对应的事件见表 1-6。

表 1-6 在 API 中打开文件

方 法	事件
Document OpenDocumentFile (string filename)	DocumentOpened
Document OpenDocumentFile (ModelPath modelPath, OpenOptions openOptions)	



第一个重载指定了某个文件的完整路径字符串后, Revit 会打开文件并创建一个 Document 实例。通过将文件通用命名转换 (UNC) 名称分配给此方法, 可以用此方法打开其他计算机上的文件。

该文件可为 .rvt 扩展名的项目文件、.rfa 扩展名的族文件或.rte 扩展名的样板文件。

第二个重载采用模型的路径作为 ModelPath 而不是字符串, OpenOptions 参数为打开该文件提供选项, 例如, 如果可以的话, 将已打开的文件从中心分离出来, 以及提供与工作共享相关的选项。打开工作共享文件的更多信息, 请参阅 5.13.3 节打开工作共享文件。

若这些方法出现故障, 会引发文件记录类的特定异常, 可分为表 1-7 中的四大类。

表 1-7 异常引发类型

类 型	异常情形
Disk errors	文件不存在或版本不正确
Resource errors	内存或硬盘空间不足, 无法打开文件
Central model file errors	文件被锁定或损坏
Central model/server errors	与服务器的网络通信错误

如果文件已成功打开, 则 DocumentOpened 事件被引发。

#### 4. 创建文件 (Create a Document)

创建新文件使用表 1-8 中的 Application 方法。

表 1-8 在 API 中创建文件

方 法	事 件
Document NewProjectDocument (string templateFileName)	DocumentCreated
Document NewFamilyDocument (string templateFileName)	DocumentCreated
Document NewProjectTemplateDocument (string templateFilename)	DocumentCreated

每个方法都需要一个样板文件的名称作为参数。方法基于此样板文件返回所创建的文件。

#### 5. 保存和关闭文件 (Save and Close a Document)

Document 类提供保存和关闭实例的方法, 其方法及事件见表 1-9。

表 1-9 在 API 中保存和关闭文件

方 法	事 件
Save()	保存文件
SaveAs()	文件保存为
Close()	关闭文件

Save()方法有两个重载, 一个不带参数, 另一个带有 SaveOptions 参数, 可以指定是否强制操作系统从硬盘文件中删除所有无用的数据。若先前从未保存过文件, 则必须调用 SaveAs()。

SaveAs()方法有三个重载, 其中之一仅含一个文件名作为参数, 若有与给定文件名重



名的其他文件存在，则引发异常。其他两个重载不仅以文件名（还有一种情况以 `ModelPath` 形式）作参数，而且以 `SaveAsOptions` 作为第二个参数，来指定文件是否重命名和/或是否覆盖现有文件。`SaveAsOptions` 参数还可用来指定其他相关选项，如是否删除硬盘上相关文件的无用数据和工作共享选项。

`Save()`、`SaveAs()`与打开文件一样有四个文件记录类的特定异常，如前述表 1-7 所列。

`Close()`方法有两个重载。其中之一带一个布尔参数以指示关闭前是否要保存该文件。第二个重载不带参数，如果文件被修改，则将询问用户在关闭之前是否要保存该文件。如果该文件的路径名称尚未设置，或者需保存的目标文件是只读的，则此方法将引发异常。

注意：`Close()`方法不会影响活动文件或引发 `DocumentClosed` 事件，因为文件由外部应用程序使用，只能对非活动文件调用此方法。

`UIDocument` 类也提供了保存和关闭实例的方法，见表 1-10。

表 1-10 在 API 中保存、关闭 `UIDocument`

方 法	事 件
<code>SaveAndClose()</code>	<code>DocumentSaved</code> , <code>DocumentClosed</code>
<code>SaveAs()</code>	<code>DocumentSavedAs</code>

`SaveAndClose()`在保存后关闭文件。如果尚未设置该文件的路径名称，将显示“`Save As`”对话框，让 Revit 用户设置其名称和位置。

`SaveAs()`方法通过“`Save As`”对话框，从 Revit 用户获得一个文件名称和路径，并以此保存文件。

## 6. 文件预览 (Document Preview)

`DocumentPreviewSettings` 类可从 `Document` 获得，其包含与给定文件预览图像储存相关的设置，见代码 1-39。

### 代码 1-39: 文件预览

```
public void SaveActiveViewWithPreview (UIApplication application)
{
    // Get the handle of current document.
    Autodesk.Revit.DB.Document document = application.ActiveUIDocument.Document;

    // Get the document's preview settings
    DocumentPreviewSettings settings = document.GetDocumentPreviewSettings();

    // Find a candidate 3D view
    FilteredElementCollector collector = new FilteredElementCollector (document);
    collector.OfClass (typeof (View3D));

    Func<View3D, bool> isValidForPreview = v => settings.IsViewIdValidForPreview (v.Id);

    View3D viewForPreview = collector.OfType<View3D>().First<View3D> (isValidForPreview);

    // Set the preview settings
```



```
Transaction setTransaction = new Transaction (document, "Set preview view id");
setTransaction.Start();
settings.PreviewViewId = viewForPreview.Id;
setTransaction.Commit();

// Save the document
document.Save();
}
```

7. 载入族 (Load Family)

Document 类可将完整的族及其所有的符号载入项目内。由于载入完整的族可能需要很长时间和大量内存，因此 Document 类提供了一个类似的方法 LoadFamilySymbol()，只载入指定的符号。

1.4.4 设置 (Settings)

表 1-11 列出了 Revit 平台用户界面管理选项卡中的命令，以及相应的 API。

表 1-11 API 及 UI 设置

用户界面命令	相关联的 API	参考
设置 ➤ 项目信息	Document.ProjectInformation	见表后注释
设置 ➤ 项目参数	Document.ParameterBindings (仅对共享参数)	见 5.1.1 节共享参数
项目位置面板	Document.ProjectLocations Document.ActiveProjectLocation	见 5.12 节地点和位置
设置 ➤ 其他设置 ➤ 填充图案	FilteredElementCollector 对 FillPatternElement 类进行过滤	见表后注释
设置 ➤ 材料	Document.Settings.Materials	见 3.9.2 节材料管理
设置 ➤ 对象样式	Document.Settings.Categories	见表后注释
阶段信息 ➤ 阶段	Document.Phases	见表后注释
设置 ➤ 结构设置	载入可用于 API 的相关结构设置	见 4.2 节 Revit Structure
设置 ➤ 项目单位	Document.GetUnits( )	见 1.4.5 节单位
面积和体积计算 (在房间&面积面板上)	Document.Settings.VoumeCalculationSetting	见表后注释

注：API 提供了可用 Document.ProjectInformation 检索的 ProjectInfo 类，来表示 Revit 项目中的项目信息。表 1-12 列出了项目信息参数对应的 API。

表 1-12 项目信息

参数	对应的 API	内建参数
Project Issue Date	ProjectInfo.IssueDate	PROJECT_ISSUE_DATE
Project Status	ProjectInfo.Status	PROJECT_STATUS
Client Name	ProjectInfo.ClientName	CLIENT_NAME
Project Address	ProjectInfo.Address	PROJECT_ADDRESS
Project Name	ProjectInfo.Name	PROJECT_NAME
Project Number	ProjectInfo.Number	PROJECT_NUMBER



使用 `ProjectInfo` 公开的属性检索并设置所有字符串。这些属性由对应的内建参数实现, 可以直接通过内建参数获取或设置字符串的值。有关如何通过内建参数访问这些参数的更多信息, 请参见 1.5 节图元概要章节中参数部分。获取项目信息的推荐方法是使用 `ProjectInfo` 属性。

- 填充图案: 使用 `FilteredElementCollector` 过滤 `FillPatternElement` 以检索当前文件中的所有 Fill Patterns。使用 `FillPatternElement.GetFillPattern (Document, ElementId)` 或 `FillPatternElement.GetFillPatternByName (Document, string)` 静态方法可以检索特定的 Fill Patterns。
- 对象样式: 使用 `Settings.Categories` 可检索 `Category` 对象中除了线型之外的所有信息。有关详细信息, 请参阅 1.5 节图元概要及 3.9 节材料章节中的类别部分。
- 阶段: Revit 用阶段来维护图元的使用期, 使用期是项目生命周期的不同时间段。使用 `Document.Phases` 属性可以检索文件中的所有阶段。该属性返回一组包含 `Phase` 类的实例。然而, Revit API 并未公开 `Phase` 类函数。
- 选项: `Options` 命令配置项目的全局设置。可以使用 `Application.Options` 属性检索 `Options.Application` 实例。目前, `Options.Application` 类只支持对库路径和共享参数文件的访问。
- 面积和体积计算: `Document.Settings.VolumeCalculationSetting` 允许启用或禁用体积计算, 以及改变房间的边界位置。

### 1.4.5 单位 (Units)

Revit API 中有关“单位”使用的两个主要的类是 `Units` 和 `FormatOptions`。`Units` 类表示将带单位的数字格式化为字符串文件的默认设置。对每个单位类型以及有关小数点和数位分组的设置, 它都包含一个 `FormatOptions` 对象。

`Units` 类为每个有效的单位类型存储一个 `FormatOptions` 对象, 但不是全部都能直接修改。某些, 像 `UT_Number` 和 `UT_SiteAngle`, 有其固定定义。另一些则有从其他单位类型自动派生出来的定义。例如, `UT_SheetLength` 是派生自 `UT_Length`, `UT_ForceScale` 是派生自 `UT_Force`。

`FormatOptions` 类包含一些设置, 控制带单位的数字如何格式化为字符串。这些设置通常是由最终用户从格式对话框中选择并存储在文件中, 例如是否舍入、精度控制, 显示单位, 是否取消空格、前导或尾随零。

`FormatOptions` 类有两种不同的使用方式。`Units` 类中的 `FormatOptions` 对象代表文件默认设置。用于其他地方的 `FormatOptions` 对象, 表示这些设置可选择性地重载默认设置。

`UseDefault` 属性控制 `FormatOptions` 对象表示默认格式还是自定义格式。如果 `UseDefault` 为 `true`, 则格式将取决于 `Units` 类中的默认设置, 对象中的其他设置均无意义。如果 `UseDefault` 为 `false`, 则该对象包含重载 `Units` 类中默认设置的自定义设置。对 `Units` 类中的 `FormatOptions` 对象, `UseDefault` 总是为 `false`。

Revit API 中与单位有重要关系的枚举包括:

- `UnitType`: 被测物理量的类型, 如长度或力 (`UT_Length` 或 `UT_Force`)。



- **DisplayUnitType**: 用于设置格式化数字为字符串或转换单位的单位和显示格式 (亦即 DUT\_METERS)。
- **UnitSymbolType**: 用格式化字符串表示数字的单位符号, 以标明值的单位 (亦即 UST\_M)。

### 1. 单位转换 (Unit Conversion)

Revit API 提供了实用程序类, 使 Revit 中的定量分类工作变得容易。UnitUtils 类可以方便地在单位数据和 Revit 内部单位之间转换。

Revit 有七个基本量, 每个都有其自己的内部单位, 这些内部单位以表 1-13 列示。

**表 1-13 Revit 单位系统中的基本单位**

基本单位	Revit 单位	单位系统
长度	英尺 (ft)	英制
角度	弧度	公制
质量	千克 (kg)	公制
时间	秒 (s)	公制
电流	安培 (A)	公制
温度	开氏度 (K)	公制
照度	坎德拉 (cd)	公制

注: 由于 Revit 以英尺存储长度, 而其他基本量采用公制单位, 因此基于英制和公制系统两者来说, 涉及长度的导出单位均为非标准单位。例如, 一个力以“质量-长度/时间的平方”来度量, 而以  $(\text{kg} \cdot \text{ft}) / \text{s}^2$  为单位来存储。

代码 1-40 使用 UnitUtils.ConvertFromInternalUnits() 方法以千磅/平方英寸为单位获取材料的屈服极限应力。

#### 代码 1-40: 从 Revit 内部单位转换

```
double GetYieldStressInKsi (Material material)
{
    double dMinYieldStress = 0;
    // Get the structural asset for the material
    ElementId strucAssetId = material.StructuralAssetId;
    if (strucAssetId != ElementId.InvalidElementId)
    {
        PropertySetElement pse = material.Document.GetElement (strucAssetId) as PropertySetElement;
        if (pse != null)
        {
            StructuralAsset asset = pse.GetStructuralAsset();

            // Get the min yield stress and convert to ksi
            dMinYieldStress = asset.MinimumYieldStress;
            dMinYieldStress = UnitUtils.ConvertFromInternalUnits (dMinYieldStress,
                DisplayUnitType.DUT_KIPS_PER_SQUARE_INCH);
        }
    }
}
```



```
return dMinYieldStress;  
}
```

UnitUtils 也可以用来将一个值从一个单位类型转换为另一单位类型,如平方英尺转换为平方米。在代码 1-41 中,以英寸输入的墙顶偏移值被转换为英尺——此值预先设置的单位。

#### 代码 1-41: 单位之间的转换

```
void SetTopOffset ( Wall wall, double dOffsetInches )  
{  
    // convert user-defined offset value to feet from inches prior to setting  
    double dOffsetFeet = UnitUtils.Convert ( dOffsetInches,  
        DisplayUnitType.DUT_DECIMAL_INCHES,  
        DisplayUnitType.DUT_DECIMAL_FEET );  
  
    Parameter paramTopOffset = wall.get_Parameter ( BuiltInParameter.WALL_TOP_OFFSET );  
    paramTopOffset.Set ( dOffsetFeet );  
}
```

## 2. 单位格式及解析 ( Unit Formatting and Parsing )

另一个实用程序类 UnitFormatUtils 可以格式化数据或解析已有格式的单位数据。

重载 FormatValueToString() 方法,用于根据格式选项将值转换为字符串,见代码 1-42。检索材料密度,然后使用 FormatValueToString() 方法,将值转换为一个用户易读的带单位的值。

#### 代码 1-42: 将值转化为字符格式

```
void DisplayDensityOfMaterial ( Material material )  
{  
    double dDensity = 0;  
    // get structural asset of material in order to get the density  
    ElementId strucAssetId = material.StructuralAssetId;  
    if ( strucAssetId != ElementId.InvalidElementId )  
    {  
        PropertySetElement pse = material.Document.GetElement ( strucAssetId ) as PropertySetElement;  
        if ( pse != null )  
        {  
            StructuralAsset asset = pse.GetStructuralAsset ( );  
  
            dDensity = asset.Density;  
            // convert the density value to a user readable string that includes the units  
            Units units = material.Document.GetUnits ( );  
            string strDensity = UnitFormatUtils.FormatValueToString ( units, UnitType.UT_UnitWeight, dDensity, false, false );  
            string msg = string.Format ( "Raw Value: {0}\r\nFormatted Value: {1}", dDensity, strDensity );  
            TaskDialog.Show ( "Material Density", msg );  
        }  
    }  
}
```

重载 UnitFormatUtils.TryParse() 方法使用指定单位类型的 Revit 内部单位,解析格式化了的字符串,包括单位,如果可能的话,将其解析成一个值。代码 1-43 获取了一个用户输



入的长度值, 假设是一个数和长度单位, 并尝试将它解析成一个长度值。将结果与任务对话框输入的字符串进行比较以说明用途。

#### 代码 1-43: 解析字符串

```
double GetLengthInput (Document document, String userInputLength)
{
    double dParsedLength = 0;
    Units units = document.GetUnits();
    // try to parse a user entered string (i.e. 100 mm, 1'6")
    bool parsed = UnitFormatUtils.TryParse (units, UnitType.UT_Length, userInputLength, out dParsedLength);
    if (parsed == true)
    {
        string msg = string.Format ("User Input: {0}\r\nParsed value: {1}", userInputLength, dParsedLength);
        TaskDialog.Show ("Parsed Data", msg);
    }

    return dParsedLength;
}
```

## 1.5 图元概要 (Elements Essentials)

图元对应于单一建筑或绘图要素, 如门、墙或尺寸。此外, 图元还可以是某个门类型、视图或材料定义。

### 1.5.1 图元分类 (Element Classification)

Revit 图元分为六组: Model (模型)、Sketch (草图)、View (视图)、Group (成组)、Annotation (注释) 和 Information (信息)。每一组都包含相关的图元和其对应的符号。

#### 1. 模型图元 (Model Elements)

模型图元表示存在于建筑项目中的物理项。模型图元组中的图元细分如下:

- **Family Instances (族实例):** 族实例包含族实例对象。可以将族对象载入项目或从族样板创建族。有关更多信息, 请参阅 3.2 节族实例。
- **Host Elements (主体图元):** 主体图元包含那些能容纳其他模型图元的系统族对象, 如墙、屋顶、天花板和楼板。关于主体图元的更多信息, 请参阅 3.1 节墙、楼板、屋顶和洞口。
- **Structure Elements (结构图元):** 结构图元只包含那些用于 Revit Structure 的图元。关于结构图元的更多信息, 请参阅 4.2 节 Revit Structure。

#### 2. 视图图元 (View Elements)

视图图元表示查看并与其他 Revit 对象进行交互的方式。有关更多信息, 请参阅 2.6 节视图。

#### 3. 成组图元 (Group Elements)

成组图元代表 Revit 中的辅助图元, 如数组和成组对象。有关详细信息, 请参阅 2.5 节编辑图元。





#### 4. 注释和基准图元 (Annotation and Datum Elements)

注释和基准图元包含可见的非物理项。

- 注释图元表示保持图纸比例的二维组件，仅在一个视图中可见。关于注释图元的更多信息，请参阅 3.6 节注释图元。

注：表示二维构件的注释图元并不仅存在于二维视图中。例如，当尺寸所参照的图形仅存在于二维平面上时，尺寸亦可在三维视图中绘制。

- 基准图元表示用来建立项目上下文的非物理项。这些图元可以存在于视图中。基准图元进一步分为以下图元：
  - Common Datum Elements (通用基准图元)：通用基准图元表示用于存储模型数据的非物理可见项。
  - Datum FamilyInstance (基准族实例)：基准族实例表示载入项目中的或从族样板创建的非物理可见项。

注：关于通用基准图元和基准族实例的更多信息，请参阅 3.5 节基准和信息图元；ModelCurve 相关内容，请参阅 3.8 节草图。

- Structural Datum Elements (结构基准图元)：结构基准图元表示用于存储结构模型数据的非物理可见项。结构基准图元的更多信息，请参阅 4.2 节 Revit Structure。

#### 5. 草图图元 (Sketch Elements)

草图图元表示二维、三维形式的草图临时项目。该组图元包含以下用于族建模和体量的对象：

- SketchPlane (草图平面)。
- Sketch (草图)。
- Path3D (三维路径)。
- GenericForm (常用形式)。

草图的详细信息，请参阅 3.8 节草绘。

#### 6. 信息图元 (Information Elements)

信息图元包含用于存储项目 and 应用程序数据的无形的非物理可见项。信息图元进一步分为以下图元：

- 项目基准图元。
- 项目基准图元 (唯一)。

关于基准图元的更多信息，请参阅 3.5 节基准和信息图元。

### 1.5.2 其他分类 (Other Classifications)

图元也可分为以下几类：

- Category (类别)。
- Family (族)。
- Symbol (符号)。
- Instance (实例)。

分类之间存在某些关系。例如：



- 可通过类别来区分不同种类的族实例。例如，结构柱项目属于结构柱类别，梁和斜撑属于结构框架类别，等等。
- 可通过符号来区分结构的族实例图元。

### 1. 类别 (Category)

**Element.Category** 属性表示图元属于某一类别或子类别。用于确定图元类型。例如，墙类别中的所有一切都被认为是墙，其他类别还包括门和房间。

类别是最一般的类，**Document.Settings.Categories** 属性是一个映射，包含文件中的所有类别对象，细分如下（参考图 1-22）：

- **Model Categories**（模型类别）：模型类别包括梁、柱、门、窗、墙。
- **Annotation Categories**（注释类别）：注释类别包括尺寸、轴网、标高和文本注释。

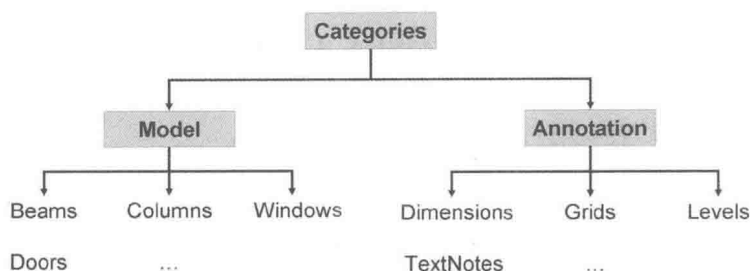


图 1-22 类别

注：类别有以下适用准则：

- 一般来说，以下规则适用于类别：
  - 每个族对象都属于某个类别。
  - 非族对象，如材料和视图，不属于类别。
  - 其他情况，如 **ProjectInfo**，属于项目信息类别。
- 图元和其相应的符号通常属于同一类别。例如，一个基本墙和通用“Generic-8”墙都属于墙类别。
- 同类型图元及相关符号也可属于不同类别。例如，**SpotDimensions** 具有 **SpotDimensionType**，但它可以属于两个不同的类别：**Spot Elevations**（点高程）和 **Spot Coordinates**（点坐标）。
- 因相似性或建筑方面的因素，不同图元也可在同一类别中。如 **ModelLine** 和 **DetailLine** 同在 **Lines** 类别中。

为了访问文件设置类中的类别（如要添加某个新的类别设置），请使用下列方法之一：

- 从文档属性获取类别，见代码 1-44。
- 用 **BuiltInCategory** 枚举从类别映射中快速获取特定类别。

#### 代码 1-44：从文件设置中获取类别

```
// Get settings of current document
Settings documentSettings = document.Settings;
```



```
// Get all categories of current document
Categories groups = documentSettings.Categories;

// Show the number of all the categories to the user
String prompt = "Number of all categories in current Revit document: " + groups.Size;

// get Floor category according to OST_Floors and show its name
Category floorCategory = groups.get_Item (BuiltInCategory.OST_Floors);
prompt += floorCategory.Name;

// Give the user some information
MessageBox.Show (prompt, "Revit", MessageBoxButtons.OK);
```

类别以下列方式使用：

- 类别用于图元归类。图元类别决定某些行为。例如，同一类别中的所有图元可列入同一明细表。
- 图元具有基于类别的参数。
- 类别还用于控制可见性和 Revit 图形外观，见图 1-23。

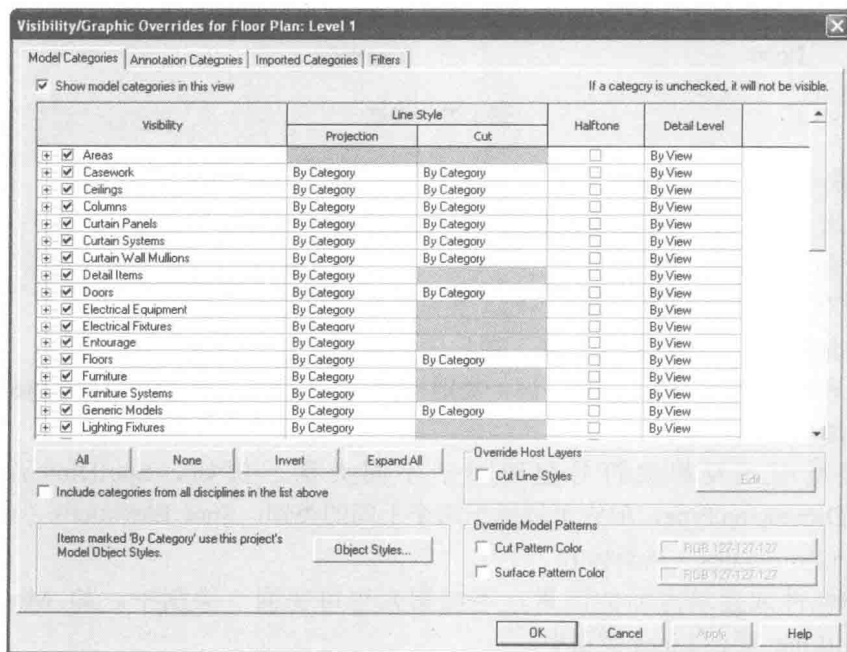


图 1-23 类别控制可见性

图元的类别由 Category ID 决定。

- 类别 ID 由 ElementId 类表示。
- 导入的类别 ID 与文件中的图元对应。
- 大多数类别是内置的，其 ID 是存储在 ElementIds 中的常数。
- 每个内置类别 ID 在 BuiltInCategory 枚举中都有一个对应的值。它们可以转换为相



应的 BuiltInCategory 枚举类型，见代码 1-45。

- 如果类别未被内置，ID 转换为空值。

#### 代码 1-45：获取图元类别

```
Element selectedElement = null;
foreach (Element e in document.Selection.Elements)
{
    selectedElement = e;
    break;    // just get one selected element
}

// Get the category instance from the Category property
Category category = selectedElement.Category;

BuiltInCategory enumCategory = (BuiltInCategory) category.Id.Value;
```

## 2. 族 (Family)

族是某类别中图元的类，族可以对图元进行以下分组：

- 一组公用参数（属性）。
- 相同的用途。
- 相似的图形表现。

大多数族都是构件族文件，这意味着可以将它们载入项目或从族样板创建，可以确定其属性设置和族的图形表现。

另一个族类型是系统族，系统族不可用于加载或创建。Revit 预定义了系统族属性和图形表现，它们包括墙、尺寸、屋顶、楼板（或板）和标高。

除了可用作图元类，族还可用作生成属于该族的新项目的样板。

在 Revit 平台 API 中，族类和族实例都属于构件族。其他图元包括系统族。

Revit 平台 API 中的族有三个对象：

- Family（族）。
- FamilySymbol（族符号）。
- FamilyInstance（族实例）。

族结构中的每个对象都起着重要作用。

族对象具有以下特点：

- 表示一个完整的族，如梁。
- 表示硬盘上一个完整的族文件。
- 包含大量的族符号。

FamilySymbol 对象表示族中一组特定的族设置，例如，类型，混凝土矩形梁：16×32。

FamilyInstance 对象是一个 FamilySymbol 实例，表示 Revit 项目中的单个实例。例如，FamilyInstance 可以是项目中 16×32 混凝土矩形剖面梁的单个实例。

注：请记住，族实例存在于族实例图元、基准图元、注释图元中。因而，适用以下规则：

- 每个族实例都有一个族符号。



- 每个族符号都属于一个族。
- 每个族都包含一个或多个族符号。

有关详细信息, 请参阅 3.2 节族实例。

### 3. 图元类型 (ElementType)

Revit 平台 API 中, 符号通常用于定义实例的不可见图元。在用户界面中符号被称为类型。

- 类型可以是某个族中的一个特定尺寸, 如一个 1730×2032 的门, 或 8×4×112 角。
- 类型可以是某个样式, 如标注用的默认的线型或角度样式。

符号表示这样一些图元, 它包含一组相似图元的共享数据。在某些情况下, 符号表示可从数据仓库获得的某些建筑构件, 如门窗, 且可以多次放置于同一建筑内。其他情况下, 符号还包含主体对象的参数或其他图元。例如, 墙类型符号包含某特定类型墙的厚度、层数、每层的材料和其他属性。

FamilySymbol 是 API 中的符号, 在 Revit 用户界面中也被称为族类型。族符号是某个族中, 所有属性值都完全相同的图元的类。例如, 所有 32×78 六面板门同属一个类型, 而所有 24×80 六面板门属于另一个类型。像族一样, 族符号也可作为样板。族符号对象派生自图元类型对象和图元对象。

### 4. 实例 (Instance)

实例是在建筑 (模型实例) 或绘图图纸 (注释实例) 中有具体位置的项目。实例表示图元类型转化为完全相同的多个副本。例如, 如果一座建筑包含 20 扇特定类型的窗, 那么该图元类型就有 20 个实例。在用户界面中实例又称为构件。

注: 对于族实例, Symbol 属性可以用来代替 GetTypeId() 方法来获得相应的族符号。这既方便又安全, 不需要做类型转换。

## 1.5.3 图元检索 (Element Retrieval)

图元在 Revit 中是很常见的。在 API 使用任何图元命令之前, 从 Revit 中检索所需的图元是必要的。用 Revit API 检索图元有以下几种方法:

- ElementId (图元 ID): 如果图元 ElementId 为已知, 则可从文件中检索该图元。
- Element Filtering and Iteration (图元过滤和迭代): 若要检索文件中一组相关图元, 这是个好方法。
- Selected Elements (选定图元): 检索用户已选定的一组图元。
- Specific Elements (特定图元): 某些图元可作为文件的属性。

所有这些图元检索方法将在后面章节分别详细讨论。

#### 1. 通过 ID 获得图元 (Getting an Element by ID)

如果所需图元的 ElementId 是已知的, 可使用 Document.Element 属性获取图元。

#### 2. 过滤图元集 (Filtering the Elements Collection)

最常见的获取文件中图元的方法, 是使用过滤来检索图元集合。Revit API 提供了 FilteredElementCollector 类及其所支持的类, 以创建可以迭代的图元过滤集。有关更多信息, 请参阅 2.1 节过滤。



### 3. 选集 (Selection)

不用获取模型中所有图元的过滤集，也可以访问已选图元。使用 `UIDocument.Selection.Elements` 属性，可以从当前活动文件中获取所选对象。有关动态选择的更多信息，请参阅 2.2 节选集。

### 4. 访问文件中特定图元 (Accessing Specific Elements from Document)

除了使用常规方法访问图元以外，还可通过 Revit 平台 API 的 `Document` 类属性从当前活动文件中来获取指定图元，而无需迭代所有图元，表 1-14 列出了可检索的特定图元。

表 1-14 从文档属性检索图元

图元	所访问的文件属性
项目信息	<code>Document.ProjectInformation</code>
项目位置	<code>Document.ProjectLocations</code> <code>Document.ActiveProjectLocation</code>
场地位置	<code>Document.SiteLocation</code>
阶段	<code>Document.Phases</code>
材料	<code>Document.Settings.Materials</code>
结构分层剖面相关族符号	<code>Document.DeckProfiles</code>
标题栏类族符号	<code>Document.TitleBlocks</code>
所有图元类型	<code>Document.AnnotationSymbolTypes / BeamSystemTypes / ContFootingTypes / DimensionTypes / FloorTypes / GridTypes / LevelTypes / RebarBarTypes / RebarHookTypes / RoomTagTypes / SpotDimensionTypes / WallTypes / TextNoteTypes</code>

## 1.5.4 通用属性 (General Properties)

以下是使用 Revit 创建的每个图元的通用属性。

### 1. 图元 ID (ElementId)

活动文件中的每个图元都有一个由 `ElementId` 存储类型表示的唯一标识符。`ElementId` 对象是项目范围内，图元模型中固定的唯一编号，这使得它可以外部存储，需要时用它来检索图元。

为了查看 Revit 图元 ID，请完成以下步骤：

- (1) 从查询面板的修改选项卡选择图元 ID，会出现图元 ID 的下拉菜单，见图 1-24。
- (2) 从选择集 IDs 中获得一个图元 ID。

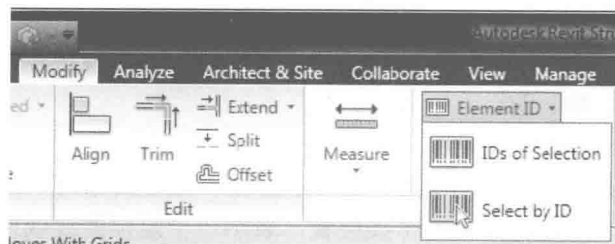


图 1-24 ElementId

在 Revit 平台 API 中，可以直接创建一个 `ElementId`，然后关联一个唯一的整数值到此



新 ElementId, 新 ElementId 默认值是 0, 见代码 1-46。

**代码 1-46: 设置 ElementId**

```
// Get the id of the element
Autodesk.Revit.DB.ElementId selectedId = element.Id;
int idInteger = selectedId.IntegerValue;

// create a new id and set the value
Autodesk.Revit.DB.ElementId id = new Autodesk.Revit.DB.ElementId (idInteger);
```

ElementId 有以下用途:

- 使用 ElementId 从 Revit 检索特定图元。从 Revit 应用程序类, 获得活动文件访问权限, 然后用 Document.GetElement (ElementId) 方法获取指定图元。见代码 1-47。

**代码 1-47: 使用 ElementId**

```
// Get the id of the element
Autodesk.Revit.DB.ElementId selectedId = element.Id;
int idInteger = selectedId.IntegerValue;

// create a new id and set the value
Autodesk.Revit.DB.ElementId id = new Autodesk.Revit.DB.ElementId (idInteger);

// Get the element
Autodesk.Revit.DB.Element first = document.GetElement (id);
```

如果项目中的 ID 不存在, 则检索的图元为 null。

- 使用 ElementId 检查一个项目中两个图元是否等同, 而不建议用 Object.Equal() 方法。

## 2. 唯一标识 (UniqueId)

每个图元都有一个字符串存储类型表示的 UniqueId, UniqueId 对应于 ElementId, 见代码 1-48。然而, 与 ElementId 不同, UniqueId 功能是个 GUID (全局唯一标识符), 跨不同 Revit 项目它也是唯一的。当将 Revit 项目文件导出为其他格式时, UniqueId 可以帮助跟踪图元。

**代码 1-48: UniqueId**

```
String uniqueId = element.UniqueId;
```

注: ElementId 仅在当前项目中是唯一的。跨不同 Revit 项目它并非唯一。而 UniqueId 跨不同的项目也始终是唯一的。

## 3. 位置 (Location)

在建模过程中, 对象的位置很重要。在 Revit 中, 某些对象有一个点位置。例如, 表格就具有一个点位置。其他对象有一个线位置, 表示一个定位曲线或根本就没有定位。墙是具有一个线位置的一种图元。

对于大多数图元, Revit 平台 API 为其提供 Location 类和定位功能。例如, 用于图元平移和旋转的 Move() 和 Rotate() 方法。然而, Location 类不能从中获取如坐标信息之类



的属性。这种情况下，可向下转换 Location 对象到其子类 LocationPoint 或 LocationCurve，以获取更详细的位置信息并使用对象派生进行控制。

获取对象的几何图形时，检索图元在项目中的物理位置是很有用的。检索位置适用以下规则：

- 墙、梁和支撑是曲线驱动的，使用 LocationCurve。
- 房间、房间标记、高程点尺寸、组、非曲线驱动的族实例，以及所有内建族实例，使用 LocationPoint。

Revit 平台 API 中，曲线驱动意味着，图元的几何图形或位置是由一个或多个关联的曲线所决定。几乎所有的分析模型图元都是曲线驱动的——线荷载和面荷载，墙、框架图元，等等。

其他图元无法检索 LocationCurve 或 LocationPoint，无法返回位置信息，详见表 1-15。

**表 1-15** 图元位置信息

位置信息	图元
LocationCurve	Wall, Beam, Brace, Structural Truss, LineLoad (无主体)
LocationPoint	Room, RoomTag, SpotDimension, Group, Column, Mass
仅 Location	Level, Floor, some Tags, BeamSystem, Rebar, Reinforcement, PointLoad, AreaLoad (无主体), Span Direction (独立标记)
无 Location	View, LineLoad (无主体), AreaLoad (无主体), BoundaryCondition

注：其他图元无位置信息，如有主体的线荷载和面荷载。

某些族实例 LocationPoint，如所有内建族实例和体量，创建时都被指定到点 (0, 0, 0)。当转换或移动该实例时，LocationPoint 坐标也随之更改。

若要更改组的 LocationPoint，请执行下列操作之一：

- 在 Revit UI 中拖曳组原点以更改 LocationPoint 坐标。在这种情况下，组的定位点改变而组的位置没有改变。
- 使用 ElementTransformUtils.MoveElement() 方法移动组来更改 LocationPoint。此方法既改变了组位置又改变了定位点。

关于定位曲线和定位点的更多信息，请参阅 2.5.1 节移动图元。

#### 4. 标高 (Level)

标高是一个有限的水平面，作为以标高为主体的或基于标高的图元（如屋顶、楼层和天花板）的参照。Revit 平台 API 提供了一个 Level 类来表示 Revit 水平线。如果图元是基于标高的，则可使用 API 获取图元所在的 Level 对象，见代码 1-49。

##### 代码 1-49：指定标高

```
//Get the level object to which the element is assigned.
Level level = element.Level;
```

许多图元，如柱，使用标高作为基面参照，见图 1-25。获取柱的标高时，检索到的标高是基面标高。



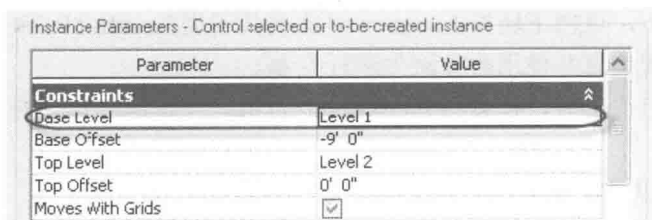


图 1-25 柱的基面标高参数

注：获取梁或斜撑标高，请使用 **Reference Level** 参数。从 **Level** 属性中你只能得到 **null**，而不是参照标高信息。

Revit 中的层是最常用的图元。在 Revit 平台 API 中，项目中的所有层通过遍历整个项目并搜索 **Elements.Level** 对象来定位。

关于标高的更多细节，请参阅 3.5 节基准和信息图元。

### 5. 参数 (Parameter)

每个图元都有一组参数，用户在 Revit 中可以查看和编辑。在图元属性对话框中，参数是可见的（任选一个图元，然后单击紧挨着类型选择器旁边的**属性**按钮）。例如，图 1-26 显示了 **Room** 参数。

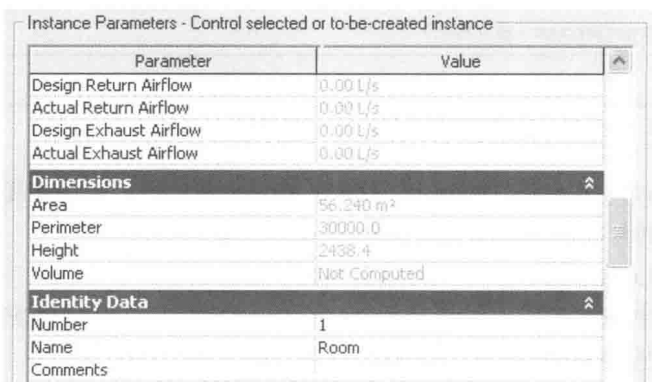


图 1-26 Room 参数

Revit 平台 API 中，每个图元对象都有参数属性，它是隶属于图元的所有属性的集合，在此集合中更改属性值。例如，从房间对象参数可以获取房间面积；另外，可以使用房间对象参数设置房间号码。参数是用来访问图元对象中未公开的属性信息的另一种方式。

通常，每个图元的参数都有一个与之关联的参数 ID。参数 ID 是由 **ElementId** 类来表示的。对用户创建的参数来说，ID 对应文件中的实际图元。然而，大多数参数是内置的，它们的 ID 是存储在 **ElementIds** 中的常数。

参数是一个存储在图元中的一般形式的数据。Revit 平台 API 中，最好使用内置参数 ID 来获取参数。使用 **BuiltInParameter** 枚举类型，可获取大量的 Revit 内置参数。

有关详细信息，请参阅 2.3 节参数。

## 第 2 章 Revit 图元基本交互

### ( Basic Interaction with Revit Elements )

#### 2.1 过滤 (Filtering)

Revit API 提供一种机制，用于过滤和迭代 Revit 文件中的图元。这是用于获取一组相关图元的最好方式，如文件中所有的墙或门。过滤器也可以用来寻找出一组很具体的图元，如某一特定尺寸的所有的梁。

通过指定过滤器获取图元的基本步骤如下：

- (1) 新建一个 `FilteredElementCollector`。
- (2) 对它运用一个或多个过滤器。
- (3) 获取滤过的图元或图元 ID (使用几种方法之一)。

代码 2-1 示例了过滤和迭代文件中图元的基本步骤。

##### 代码 2-1：使用图元过滤得到文件中的所有墙

```
// Find all Wall instances in the document by using category filter
ElementCategoryFilter filter = new ElementCategoryFilter (BuiltInCategory.OST_Walls);

// Apply the filter to the elements in the active document
// Use shortcut WhereElementsIsNotElementType() to find wall instances only
FilteredElementCollector collector = new FilteredElementCollector (document);
IList<Element> walls =
collector.WherePasses (filter).WhereElementsIsNotElementType().ToElements();
String prompt = "The walls in the current document are: \n";
foreach (Element e in walls)
{
    prompt += e.Name + "\n";
}
TaskDialog.Show ("Revit", prompt);
```

##### 2.1.1 创建图元过滤集 (Create a FilteredElementCollector)

用于图元的迭代和过滤的主类称作 `FilteredElementCollector`。可用以下三种方式之一来建立：

- (1) 文件。搜索并过滤文件中的图元集。
- (2) 文件和图元 ID 集。搜索并过滤文件中的指定图元集。
- (3) 文件和视图。搜索并过滤视图中的可见图元。

注：使用静态方法 `FilteredElementCollector.IsViewValidForElementIteration()` 在某特定



视图中过滤图元时，总是要检查视图对图元迭代的有效性。

当对象为首次创建时，无适用的过滤器。在尝试访问图元之前，此类要求至少设置一个条件，否则将引发异常。

### 2.1.2 应用过滤器 ( Applying Filters )

可以用 `ElementFilters` 对 `FilteredElementCollector` 使用过滤器。`ElementFilter` 是一个类，用于检查图元是否满足某种条件。`ElementFilter` 基类有三个派生类，将图元过滤器划分为三类。

- `ElementQuickFilter`。快速过滤器仅对 `ElementRecord` 进行操作，是一个低内存占用的类，以一个有限接口来读取图元属性。被快速过滤器丢弃的图元不会展开到内存中。
- `ElementSlowFilter`。慢速过滤器首先需要获取图元并展开到内存中。因此，更为可取的方法是，将慢速过滤器与至少一个快速过滤器结合使用，尽量减少展开到内存的图元数量，以对照此过滤器设置的标准进行评价。
- `ElementLogicalFilter`。逻辑过滤器逻辑组合两个或更多过滤器。`Revit` 以使过滤器执行最快为优先评估条件，可能会将合成过滤器重新排序。

带一个指示过滤器反转的布尔参数使用重载构造函数，大多数过滤器都可反转使用，以使正常能为过滤器所接受的图元被拒，正常被拒的图元被接受。那些不能反转的过滤器，后面相应部分会作注解。

有一组预定义的常用过滤器。这些内置的过滤器为前述 `FilteredElementCollector` 部分提及的 `FilteredElementCollector` 快捷方法提供基础。

过滤器创建之后，即需应用到 `FilteredElementCollector` 中。泛型方法 `WherePasses()` 用于对 `FilteredElementCollector` 使用单个 `ElementFilter`。

也可以用 `FilteredElementCollector` 提供的一些快捷方法来使用过滤器。其中一些方法应用一个没有进一步输入的特定过滤器，如 `WhereElementIsCurveDriven()`，而其他一些方法应用有一个简单输入的特定过滤器，如 `OfCategory()` 方法以 `BuiltInCategory` 作为参数，还有一些方法如 `UnionWith()`，可将几个过滤器结合在一起。所有这些方法都返回同一采集器，让过滤器可以方便地连接在一起。

#### 1. 快速过滤器 ( Quick Filters )

快速过滤器仅对 `ElementRecord` 进行操作，是一个低内存占用的类，以一个有限接口来读取图元属性。被快速过滤器丢弃的图元不会展开到内存中。表 2-1 概括了内置快速过滤器，代码 2-2~代码 2-4 是过滤器的一些例子。

表 2-1 内置快速过滤器

内置过滤器	过滤结果	快捷方法
<code>BoundingBoxContainsPointFilter</code>	包含一个给定点的有边界框的图元	无
<code>BoundingBoxIntersectsFilter</code>	有一个与给定轮廓相交的边界框的图元	无
<code>BoundingBoxIsInsideFilter</code>	有一个边界框在给定轮廓内的图元	无
<code>ElementCategoryFilter</code>	与输入类别 ID 匹配的图元	<code>OfCategoryId()</code>



续表

内置过滤器	过滤结果	快捷方法
ElementClassFilter	与输入的运行类 (或派生类) 匹配的图元	OfClass( )
ElementDesignOptionFilter	符合一个特定的设计选项的图元	ContainedInDesignOption( )
ElementIsCurveDrivenFilter	曲线驱动的图元	WhereElementIsCurveDriven( )
ElementIsElementTypeFilter	是一个“图元类型”的图元	WhereElementIsElementType( )、 WhereElementIsNotElementType( )
ElementMulticategoryFilter	与任何一组给定的类别匹配的图元	无
ElementMulticlassFilter	与一组给定的类别 (或派生类) 匹配的图元	无
ElementOwnerViewFilter	有专有视图的图元	OwnedByView( )、 WhereElementIsViewIndependent( )
ElementStructuralTypeFilter	与给定的结构类型匹配的图元	无
ExclusionFilter	除图元 ID 被输入过滤器以外的全部图元	Excluding( )
FamilySymbolFilter	特定族符号	

注 1. FamilySymbolFilter 不能反转。

2. 边界框过滤器不包括所有从视图和图元类型派生的对象。

代码 2-2 示例了在文件中创建一个轮廓, 使用 BoundingBoxIntersectsFilter 过滤器得到所有边界框与该轮廓相交的图元。然后演示了如何使用一个反向过滤器找出所有的墙, 其边界框与给定的轮廓不相交。注意 OfClass() 方法的用途, 还可将 ElementClassFilter 应用于集合。

#### 代码 2-2: Bounding Box Intersects Filter 实例

```
// Use BoundingBoxIntersects filter to find elements with a bounding box that intersects the
// given Outline in the document.

// Create a Outline, uses a minimum and maximum XYZ point to initialize the outline.
Outline myOutLn = new Outline (new XYZ (0, 0, 0), new XYZ (100, 100, 100));

// Create a BoundingBoxIntersects filter with this Outline
BoundingBoxIntersectsFilter filter = new BoundingBoxIntersectsFilter (myOutLn);

// Apply the filter to the elements in the active document
// This filter excludes all objects derived from View and objects derived from ElementType
FilteredElementCollector collector = new FilteredElementCollector (document);
IList<Element> elements = collector.WherePasses (filter) .ToElements();

// Find all walls which don't intersect with BoundingBox: use an inverted filter
// to match elements
// Use shortcut command OfClass( ) to find walls only
BoundingBoxIntersectsFilter invertFilter = new BoundingBoxIntersectsFilter (myOutLn, true);
collector = new FilteredElementCollector (document);
IList<Element> notIntersectWalls = collector.OfClass (typeof (Wall)) .WherePasses (invertFilter) .ToElements();
```

代码 2-3 使用一个排他过滤器, 查找文件中当前未选定的所有墙。

**代码 2-3: 创建一个排他过滤器**

```
// Find all walls that are not currently selected,
// Get all element ids which are current selected by users, exclude these ids when filtering
ICollection<ElementId> excludes = new List<ElementId>();
ElementSetIterator elemIter = uiDocument.Selection.Elements.ForwardIterator();
elemIter.Reset();
while (elemIter.MoveNext())
{
    Element curElem = elemIter.Current as Element;
    excludes.Add (curElem.Id);
}

// Create filter to exclude all selected element ids
ExclusionFilter filter = new ExclusionFilter (excludes);

// Apply the filter to the elements in the active document,
// Use shortcut method OfClass() to find Walls only
FilteredElementCollector collector = new FilteredElementCollector (uiDocument.Document);
IList<Element> walls = collector.WherePasses (filter).OfClass (typeof (Wall)).ToElements();
```

注意: `ElementClassFilter` 将匹配这样的图元, 它们的类与输入的类准确匹配, 或这些图元的类是输入类的派生类。代码 2-4 使用 `ElementClassFilter` 获取文件中的全部荷载。

**代码 2-4: 用 `ElementClassFilter` 获取荷载**

```
// Use ElementClassFilter to find all loads in the document
// Using typeof (LoadBase) will yield all AreaLoad, LineLoad and PointLoad
ElementClassFilter filter = new ElementClassFilter (typeof (LoadBase));

// Apply the filter to the elements in the active document
FilteredElementCollector collector = new FilteredElementCollector (document);
ICollection<Element> allLoads = collector.WherePasses (filter).ToElements();
```

在 API 中存在图元类过滤器不支持的图元子类类型, 这些类型存在于 API 中, 但 Revit 原生对象模型中不包括它们, 这意味着过滤器不支持它们。为了使用类过滤器来找到这些类型的图元, 需使用更高层级的类, 然后进一步处理以找出只匹配该子类型的图元。

请注意, 其中的某些类型有其专用过滤器。受此限制影响的类型见表 2-2。

表 2-2 类型专用过滤器

类 型	专用过滤器
Subclasses of Autodesk.Revit.DB.Material	无
Subclasses of Autodesk.Revit.DB.CurveElement	CurveElementFilter
Subclasses of Autodesk.Revit.DB.ConnectorElement	无
Subclasses of Autodesk.Revit.DB.HostedSweep	无
Autodesk.Revit.DB.Architecture.Room	RoomFilter
Autodesk.Revit.DB.Mechanical.Space	SpaceFilter

续表

类 型	专用过滤器
Autodesk.Revit.DB.Area	AreaFilter
Autodesk.Revit.DB.Architecture.RoomTag	RoomTagFilter
Autodesk.Revit.DB.Mechanical.SpaceTag	SpaceTagFilter
Autodesk.Revit.DB.AreaTag	AreaTagFilter
Autodesk.Revit.DB.CombinableElement	无
Autodesk.Revit.DB.Mullion	无
Autodesk.Revit.DB.Panel	无
Autodesk.Revit.DB.AnnotationSymbol	无
Autodesk.Revit.DB.Structure.AreaReinforcementType	无
Autodesk.Revit.DB.Structure.PathReinforcementType	无
Autodesk.Revit.DB.AnnotationSymbolType	无
Autodesk.Revit.DB.Architecture.RoomTagType	无
Autodesk.Revit.DB.Mechanical.SpaceTagType	无
Autodesk.Revit.DB.AreaTagType	无
Autodesk.Revit.DB.Structure.TrussType	无

2. 慢速过滤器 (Slow Filters)

慢速过滤器需要获取图元并展开到内存中。因此，更为可取的方法是，将慢速过滤器与至少一个快速过滤器结合使用，尽量减少展开到内存的图元数量。表 2-3 概括了这些内置慢速过滤器，而通过代码 2-5~代码 2-7，可对某些过滤器加深了解。

表 2-3 内置慢速过滤器

内置过滤器	过滤结果	快捷方法
AreaFilter	面积	无
AreaTagFilter	面积标记	无
CurveElementFilter	曲线图元	无
ElementLevelFilter	与给定标高 ID 相关的图元	无
ElementParameterFilter	符合一个或多个过滤规则的图元	无
ElementPhaseStatusFilter	对应给定阶段有给定阶段状态的图元	无
FamilyInstanceFilter	特定族实例的实例	无
FamilyStructuralMaterialTypeFilter	给定结构材料类型的族图元	无
PrimaryDesignOptionMemberFilter	属于任何初步设计选项的图元	无
RoomFilter	房间	无
RoomTagFilter	房间标记	无
SpaceFilter	空间	无
SpaceTagFilter	空间标记	无



续表

内置过滤器	过滤结果	快捷方法
StructuralInstanceUsageFilter	给定结构用法的族实例	无
StructuralMaterialTypeFilter	给定结构材料类型的族实例	无
StructuralWallUsageFilter	给定结构墙用法的墙	无
ElementIntersectsElementFilter	与给定图元几何实体相交的图元	无
ElementIntersectsSolidFilter	与给定几何实体相交的图元	无

以下慢速过滤器不能反转：

- RoomFilter。
- RoomTagFilter。
- AreaFilter。
- AreaTagFilter。
- SpaceFilter。
- SpaceTagFilter。
- FamilyInstanceFilter。

在快速过滤器部分中提到，ElementClassFilter 无法操作某些类。其中的某些类，如房间和房间标记有其专用过滤器，代码 2-5 示例了 Room 专用过滤器。

#### 代码 2-5： 使用 Room 过滤器

```
// Use a RoomFilter to find all room elements in the document. It is necessary to use the
// RoomFilter and not an ElementClassFilter or the shortcut method OfClass() because the Room
// class is not supported by those methods.
RoomFilter filter = new RoomFilter();

// Apply the filter to the elements in the active document
FilteredElementCollector collector = new FilteredElementCollector (document);
IList<Element> rooms = collector.WherePasses (filter) .ToElements();
```

ElementParameterFilter 是一个功能强大的过滤器，可以根据其参数值找出图元。它可以找出这些图元，其参数值与特定值相匹配或大于、小于某个值。ElementParameterFilter 还可用来找出支持特定共享参数的图元。

代码 2-6 示例了使用 ElementParameterFilter 找出面积大于 100 平方英尺的房间，以及小于 100 平方英尺的房间。

#### 代码 2-6： 使用参数过滤器

```
// Creates an ElementParameter filter to find rooms whose area is
// greater than specified value
// Create filter by provider and evaluator
BuiltInParameter areaParam = BuiltInParameter.ROOM_AREA;
// provider
ParameterValueProvider pvp = new ParameterValueProvider (new ElementId ((int) areaParam));
// evaluator
```



```

FilterNumericRuleEvaluator fnrv = new FilterNumericGreater();
// rule value
double ruleValue = 100.0f; // filter room whose area is greater than 100 SF
// rule
FilterRule fRule = new FilterDoubleRule (pvp, fnrv, ruleValue, 1E-6);

// Create an ElementParameter filter
ElementParameterFilter filter = new ElementParameterFilter (fRule);

// Apply the filter to the elements in the active document
FilteredElementCollector collector = new FilteredElementCollector (document);
IList<Element> rooms = collector.WherePasses (filter) .ToElements();

// Find rooms whose area is less than or equal to 100:
// Use inverted filter to match elements
ElementParameterFilter lessOrEqualFilter = new ElementParameterFilter (fRule, true);
collector = new FilteredElementCollector (document);
IList<Element> lessOrEqualFounds = collector.WherePasses (lessOrEqualFilter) .ToElements();

```

代码 2-7 演示了如何使用 `FamilyStructuralMaterialTypeFilter` 找出所有材料类型是木材的族。它还演示了如何使用一个反转过滤器来查出所有材料类型不是木材的族。

#### 代码 2-7：找出材料为木材的族

```

// Use FamilyStructuralMaterialType filter to find families whose material type is Wood
FamilyStructuralMaterialTypeFilter filter = new FamilyStructuralMaterialTypeFilter (StructuralMaterialType.Wood);

// Apply the filter to the elements in the active document
FilteredElementCollector collector = new FilteredElementCollector (document);
ICollection<Element> woodFamilies = collector.WherePasses (filter) .ToElements();

// Find families are not Wood: Use inverted filter to match families
FamilyStructuralMaterialTypeFilter notWoodFilter =
    new FamilyStructuralMaterialTypeFilter (StructuralMaterialType.Wood, true);
collector = new FilteredElementCollector (document);
ICollection<Element> notWoodFamilies = collector.WherePasses (notWoodFilter) .ToElements();

```

`ElementIntersectsElementFilter` 和 `ElementIntersectsSolidFilter` 两个慢速过滤器由 `ElementIntersectsFilter` 基类派生，该过滤器的基类用于匹配与几何形体相交的图元。

### 3. 逻辑过滤器 (Logical Filters)

逻辑过滤器逻辑组合两个或更多过滤器。表 2-4 概括了内置的逻辑过滤器。

表 2-4 内置逻辑过滤器

内置过滤器	过滤结果	快捷方法
<code>LogicalAndFilter</code>	通过多组过滤器的图元	<code>WherePasses()</code> : 添加一个附加过滤器 <code>IntersectWith()</code> : 连接两组独立过滤器
<code>LogicalOrFilter</code>	多组过滤器中，至少通过一组过滤器的图元	<code>UnionWith()</code> : 连接两组独立过滤器





在代码 2-8 实例中, 两个快速过滤器组合成逻辑过滤器, 用来获取文件中的所有门族实例图元。

#### 代码 2-8: 用 LogicalAndFilter 找出所有的门

```
// Find all door instances in the project by finding all elements that both belong to the
// door category and are family instances.
ElementClassFilter familyInstanceFilter = new ElementClassFilter (typeof (FamilyInstance));

// Create a category filter for Doors
ElementCategoryFilter doorsCategoryfilter =
    new ElementCategoryFilter (BuiltInCategory.OST_Doors);

// Create a logic And filter for all Door FamilyInstances
LogicalAndFilter doorInstancesFilter = new LogicalAndFilter (familyInstanceFilter,
    doorsCategoryfilter);

// Apply the filter to the elements in the active document
FilteredElementCollector collector = new FilteredElementCollector (document);
IList<Element> doors = collector.WherePasses (doorInstancesFilter).ToElements();
```

### 2.1.3 获取过滤图元或图元 ID (Getting Filtered Elements or Element IDs)

对 FilteredElementCollector 应用一个或多个过滤器之后, 过滤出的图元集可由以下三种方式之一来检索:

(1) 获取图元或图元 ID 的集合。

- ToElements(): 返回通过过滤器的所有图元。
- ToElementIds(): 返回通过过滤器所有图元的图元 ID。

(2) 获取匹配过滤器的第一个图元或图元 ID。

- FirstElement(): 返回通过过滤器的第一个图元。
- FirstElementId(): 返回通过过滤器第一个图元的 ID。

(3) 获取图元 ID 或图元迭代器。

- GetElementIdIterator(): 返回通过过滤器的图元 ID 的 FilteredElementIdIterator。
- GetElementIterator(): 返回通过过滤器的图元的 FilteredElementIdIterator。
- GetEnumerator(): 返回一个 IEnumerator<Element>枚举, 其遍历通过的图元集。

每次应只使用这几组方法其中之一; 如果再调用其他方法来提取图元, 采集器会重置。因而, 如果先前已获得一个迭代器, 若再调用其他方法来提取图元, 则会被中止, 而不会遍历更多图元。

如果仅需得到一个匹配图元, 则 FirstElement()或 FirstElementId()是最好的选择。如果需获得所有匹配的图元, 则使用 ToElements()。如果要得到的图元数量是可变的, 则使用迭代器。如果应用程序要删除过滤列表中的图元或对其作重大更改, 则使用 ToElementIds()或图元 ID 迭代器是最好的选择。这是因为删除图元或图元作出重大修改, 会使图元句柄失效。有了图元 ID, 带 ElementId 参数调用 Document.GetElement(), 则始终会返回一个有效的图元 (如果图元已被删除, 则返回一个空引用)。



利用 `ToElements()` 方法来获取过滤结果作为一个图元集合, 允许使用 `foreach` 遍历图元集合中的每个图元, 见代码 2-9。

#### 代码 2-9: 用 `ToElements()` 获取过滤结果

```
// Use ElementClassFilter to find all loads in the document
// Using typeof (LoadBase) will yield all AreaLoad, LineLoad and PointLoad
ElementClassFilter filter = new ElementClassFilter (typeof (LoadBase));

// Apply the filter to the elements in the active document
FilteredElementCollector collector = new FilteredElementCollector (document);
collector.WherePasses (filter);
ICollection<Element> allLoads = collector.ToElements();

String prompt = "The loads in the current document are: \n";
foreach (Element loadElem in allLoads)
{
    LoadBase load = loadElem as LoadBase;
    prompt += load.GetType().Name + ": " +
        load.Name + "\n";
}

TaskDialog.Show ("Revit", prompt);
```

当只需要一个通过的图元时, 使用 `FirstElement()`, 见代码 2-10。

#### 代码 2-10: 获取通过的第一个图元

```
// Create a filter to find all columns
StructuralInstanceUsageFilter columnFilter =
    new StructuralInstanceUsageFilter (StructuralInstanceUsage.Column);

// Apply the filter to the elements in the active document
FilteredElementCollector collector = new FilteredElementCollector (document);
collector.WherePasses (columnFilter);

// Get the first column from the filtered results
// Element will be a FamilyInstance
FamilyInstance column = collector.FirstElement() as FamilyInstance;
```

在某些情况下, 使用 `FirstElement()` 是不够的。代码 2-11 演示了如何使用扩展方法来获取第一个非样板三维视图 (可用于 `ReferenceIntersector` 构造函数的输入)。

#### 代码 2-11: 用扩展方法获取第一个通过的图元

```
// Use filter to find a non-template 3D view
// This example does not use FirstElement() since first filtered view3D might be a template
FilteredElementCollector collector = new FilteredElementCollector (document);
Func<View3D, bool> isNotTemplate = v3 => ! (v3.IsTemplate);

// apply ElementClassFilter
```



```
collector.OfClass (typeof (View3D));

// use extension methods to get first non-template View3D
View3D view3D = collector.Cast<View3D>().First<View3D> (isNotTemplate);
```

代码 2-12 演示了如何使用 `FirstElementId()` 方法来获取一个通过的图元 (本例为三维视图), 并用 `ToElementIds()` 获取图元 ID 集合的过滤结果 (本例目的是为了删除一组图元)。

#### 代码 2-12: 使用获取图元 ID 过滤结果

```
FilteredElementCollector collector = new FilteredElementCollector (document);

// Use shortcut OfClass to get View elements
collector.OfClass (typeof (View3D));

// Get the Id of the first view
ElementId viewId = collector.FirstElementId();

// Test if the view is valid for element filtering
if (FilteredElementCollector.IsViewValidForElementIteration (document, viewId))
{
    FilteredElementCollector viewCollector = new FilteredElementCollector (document, viewId);

    // Get all FamilyInstance items in the view
    viewCollector.OfClass (typeof (FamilyInstance));
    ICollection<ElementId> familyInstanceIds = viewCollector.ToElementIds();

    document.Delete (familyInstanceIds);
}
```

代码 2-13 使用 `GetElementIterator()` 方法遍历过滤图元来检查某些管道的流动状态。

#### 代码 2-13: 获取结果用作图元迭代器

```
FilteredElementCollector collector = new FilteredElementCollector (document);

// Apply a filter to get all pipes in the document
collector.OfClass (typeof (Autodesk.Revit.DB.Plumbing.Pipe));

// Get results as an element iterator and look for a pipe with
// a specific flow state
FilteredElementIterator elemItr = collector.GetElementIterator();
elemItr.Reset();
while (elemItr.MoveNext())
{
    Pipe pipe = elemItr.Current as Pipe;
    if (pipe.FlowState == PipeFlowState.LaminarState)
    {
        TaskDialog.Show ("Revit", "Model has at least one pipe with Laminar flow state.");
        break;
    }
}
```



另外, 过滤结果也可以返回为图元 ID 迭代器, 见代码 2-14。

#### 代码 2-14: 获取图元 ID 迭代结果

```
// Use a RoomFilter to find all room elements in the document.
RoomFilter filter = new RoomFilter();

// Apply the filter to the elements in the active document
FilteredElementCollector collector = new FilteredElementCollector (document);
collector.WherePasses (filter);

// Get results as ElementId iterator
FilteredElementIdIterator roomIdItr = collector.GetElementIdIterator();
roomIdItr.Reset();
while (roomIdItr.MoveNext())
{
    ElementId roomId = roomIdItr.Current;
    // Warn rooms smaller than 50 SF
    Room room = document.GetElement (roomId) as Room;
    if (room.Area < 50.0)
    {
        String prompt = "Room is too small: id = " + roomId.ToString();
        TaskDialog.Show ("Revit", prompt);
        break;
    }
}
```

在某些情况下, 只想得到通过过滤器的特定图元, 而不是获取所有通过过滤器的图元。ElementFilter.PassesFilter() 有两个重载, 根据过滤器检索给定的图元或图元 ID, 若图元在过滤器中满足检索条件, 则返回 true。

#### 2.1.4 LINQ 查询 (LINQ Queries)

在 .NET 中, FilteredElementCollector 类支持图元的 IEnumerable 接口。可以通过 LINQ 查询使用该类以及处理图元列表的操作。注意, 由于 ElementFilters 和该类提供的快捷方法, 在它们的托管封装器生成之前以本机代码处理图元, 因此, 在尝试用 LINQ 查询处理结果之前, 在采集器上尽可能多地使用本地过滤器将会获得更好的性能。

代码 2-15 示例了用 ElementClassFilter 获得文件中的所有族实例图元, 然后使用 LINQ 查询以缩小那些具有特定名称的族实例结果的范围。

#### 代码 2-15: 使用 LINQ 查询

```
// Use ElementClassFilter to find family instances whose name is 60" x 30" Student
ElementClassFilter filter = new ElementClassFilter (typeof (FamilyInstance));

// Apply the filter to the elements in the active document
FilteredElementCollector collector = new FilteredElementCollector (document);
collector.WherePasses (filter);
```



```
// Use Linq query to find family instances whose name is 60" x 30" Student
var query = from element in collector
            where element.Name == "60\" x 30\" Student"
            select element;

// Cast found elements to family instances,
// this cast to FamilyInstance is safe because ElementClassFilter for FamilyInstance was used
List<FamilyInstance> familyInstances = query.Cast<FamilyInstance>( ).ToList<FamilyInstance>( );
```

## 2.1.5 边界框过滤器 ( Bounding Box Filters )

边界框过滤器:

- BoundingBoxIsInsideFilter。
- BoundingBoxIntersectsFilter。
- BoundingBoxContainsPointFilter。

边框过滤器可以找出具有符合特定条件边界框的图元, 检查每个图元的边界框是否在给定体量内, 是否与给定体量相交或包含一个给定的点。这一检查也可以反转使用, 以找到与给定体量不发生交集或不包含一个给定点的图元。

BoundingBox 过滤器使用轮廓作为其输入。Outline 是一个类, 表示一个与 Revit 全局坐标系轴对齐的长方体轮廓。

边界框过滤器对于图元实际几何形状与它的边界框几何形状紧密匹配过滤效果好, 如包括与  $X$  或  $Y$  方向平行的线性墙、由这些墙围成的矩形房间、与这些墙对齐的楼板或屋顶, 或适当的矩形族; 否则, 当图元的边界框比实际几何形状大太多时, 有可能出现误报 (在这些情况下, 可以使用实际的图元几何形状确定图元是否真的符合标准)。

## 2.1.6 图元相交过滤器 ( Element Intersection Filters )

图元过滤器如下:

- ElementIntersectsElementFilter。
- ElementIntersectsSolidFilter。

图元过滤器过滤实际三维几何形状与目标对象三维几何形状相交的图元。

使用 ElementIntersectsElementFilter, 其目标对象是另一个图元。是否存在交集, 与 Revit 在干扰报告生成期间, 与 Revit 在碰撞检查报告生成时确定是否存在冲突有同样的判断逻辑 (这意味着一些图元组合将不会通过此过滤器, 如自动加入其交集的混凝土构件)。此外, 不具有几何实体的图元, 如钢筋, 也不会通过此过滤器。

使用 ElementIntersectsSolidFilter, 其目标对象是任何实体。该实体可以从现有图元获取的, 也可使用 GeometryCreationUtilities 创建, 或使用二次操作如布尔运算的结果。类似于 ElementIntersectsElementFilter, 该过滤器将不会通过缺乏几何实体的图元。

这两个过滤器可以反转匹配在目标对象体之外的图元。

这两个过滤器均为慢速过滤器, 因此最好是与一个或多个快速过滤器 (如类或类别过滤器) 结合使用, 见代码 2-16。



代码 2-16: 使用 ElementIntersectsSolidFilter 筛选图元

```

/// <summary>
/// Finds any Revit physical elements which interfere with the target
/// solid region surrounding a door.</summary>
/// <remarks>This routine is useful for detecting interferences which are
/// violations of the Americans with Disabilities Act or other local disabled
/// access codes.</remarks>
/// <param name="doorInstance">The door instance.</param>
/// <param name="doorAccessibilityRegion">The accessibility region calculated
/// to surround the approach of the door.
/// Because the geometric parameters of this region are code- and
/// door-specific, calculation of the geometry of the region is not
/// demonstrated in this example.</param>
/// <returns>A collection of interfering element ids.</returns>
private ICollection<ElementId> FindElementsInterferingWithDoor (FamilyInstance doorInstance, Solid doorAccessibilityRegion)
{
    // Setup the filtered element collector for all document elements.
    FilteredElementCollector interferingCollector =
        new FilteredElementCollector (doorInstance.Document);

    // Only accept element instances
    interferingCollector.WhereElementIsNotElementType();

    // Exclude intersections with the door itself or the host wall for the door.
    List<ElementId> excludedElements = new List<ElementId>();
    excludedElements.Add (doorInstance.Id);
    excludedElements.Add (doorInstance.Host.Id);
    ExclusionFilter exclusionFilter = new ExclusionFilter (excludedElements);
    interferingCollector.WherePasses (exclusionFilter);

    // Set up a filter which matches elements whose solid geometry intersects
    // with the accessibility region
    ElementIntersectsSolidFilter intersectionFilter =
        new ElementIntersectsSolidFilter (doorAccessibilityRegion);
    interferingCollector.WherePasses (intersectionFilter);

    // Return all elements passing the collector
    return interferingCollector.ToElementIds();
}

```

## 2.2 选集 (Selection)

使用 `UIDocument.Selection.Elements` 属性, 可从当前活动文件获取所选对象。所选对象在 `Revit ElementSet` 中。所有已选取的图元都从这个图元集中检索。选集对象还可通过编程方式更改当前所选内容。

作为另一个选择, `Selection.GetElementIds()` 方法与 `Selection.Elements` 属性一样可



检索同样的图元集。此方法返回的集合可直接用于 `FilteredElementCollector` 过滤所选图元。

### 2.2.1 更改选集 (Changing the Selection)

为了修改 `Selection.Elements`:

- (1) 新建一个 `SelElementSet`。
- (2) 将图元放入。
- (3) 设置 `Selection.Elements` 到新建的 `SelElementSet` 实例。

代码 2-17 说明了如何更改已选定的图元。

代码 2-17: 更改已选定的图元

```
private void ChangeSelection (Document document)
{
    // Get selected elements form current document.
    UIDocument uidoc = new UIDocument (document);
    Autodesk.Revit.UI.Selection.SelElementSet collection = uidoc.Selection.Elements;

    // Display current number of selected elements
    TaskDialog.Show ("Revit", "Number of selected elements: " + collection.Size.ToString());

    //Create a new SelElementSet
    SelElementSet newSelectedElementSet = SelElementSet.Create();

    // Add wall into the created element set.
    foreach (Autodesk.Revit.DB.Element elements in collection)
    {
        if (elements is Wall)
        {
            newSelectedElementSet.Add (elements);
        }
    }

    // Set the created element set as current select element set.
    uidoc.Selection.Elements = newSelectedElementSet;

    // Give the user some information.
    if (0 != newSelectedElementSet.Size)
    {
        TaskDialog.Show ("Revit", uidoc.Selection.Elements.Size.ToString() +
            " Walls are selected!");
    }
    else
    {
        TaskDialog.Show ("Revit", "No Walls have been selected!");
    }
}
```



### 2.2.2 用户选集 (User Selection)

Selection 类还有一些允许用户选择新对象, 甚至屏幕上一个点的方法。这让用户可以使用光标选择一个或多个图元 (或其他对象, 如边或面), 然后将控制返回给应用程序。这些功能并不自动向活动选择集添加新的选择。

- PickObject()方法提示用户选择一个 Revit 模型中的对象。
- PickObjects()方法提示用户选择多个 Revit 模型中的对象。
- PickElementsByRectangle()方法提示用户用矩形选择多个 Revit 模型中的对象。
- PickPoint()方法提示用户在活动草图平面内拾取一个点。
- PickBox()方法调用一个通用的双击编辑器, 让用户在屏幕上指定一个矩形区域。

调用 PickObject()或 PickObjects()时即指定了需选对象的类型。可指定的对象类型有图元和图元上的点、边或面。

每个 Pick 函数都可被重载, 重载时可以带一个字符串参数, 该参数用于定制状态栏消息, 见代码 2-18, 当开发人员的应用程序提示用户拾取对象或图元时, 该消息会在状态栏上显示。

**代码 2-18: 用 PickObject()和 PickElementsByRectangle()添加所选图元**

```
UIDocument uidoc = new UIDocument (document);
Selection choices = uidoc.Selection;
// Pick one object from Revit.
Reference hasPickOne = choices.PickObject (ObjectType.Element);
if (hasPickOne != null)
{
    TaskDialog.Show ("Revit", "One element added to Selection.");
}

int selectionCount = choices.Elements.Size;
// Choose objects from Revit.
IList<Element> hasPickSome = choices.PickElementsByRectangle ("Select by rectangle");
if (hasPickSome.Count > 0)
{
    int newSelectionCount = choices.Elements.Size;
    string prompt = string.Format ("{0} elements added to Selection.",
        newSelectionCount - selectionCount);
    TaskDialog.Show ("Revit", prompt);
}
```

PickPoint()方法有两个带有 ObjectSnapTypes 参数的重载, 该参数可指定供选择用的捕捉类型。可以指定多个捕捉类型, 见代码 2-19。

**代码 2-19: 捕捉点**

```
public void PickPoint (UIDocument uidoc)
{
    ObjectSnapTypes snapTypes = ObjectSnapTypes.Endpoints | ObjectSnapTypes.Intersections;
    XYZ point = uidoc.Selection.PickPoint (snapTypes, "Select an end point or intersection");
}
```





```
string strCoords = "Selected point is " + point.ToString();

TaskDialog.Show ("Revit", strCoords);

}
```

PickBox()方法采用 PickBoxStyle 枚举器。选项为: Crossing (交叉), 此样式用于所选对象全部或局部在框内; Enclosing (围合), 此样式用于所选对象完全被框所围合; Directional (定向), 框的样式取决于正在绘制框的方向。如果它正从右到左化绘制, 使用 Crossing 样式; 相反则使用 Enclosing 样式。

PickBox()返回一个包含最小和最大选中点的 PickedBox。代码 2-20 演示了在 Point Cloud (点云) 选集中如何使用 PickBox()。

#### 代码 2-20: 选择框

```
public void PromptForPointCloudSelection (UIDocument uiDoc, PointCloudInstance pcInstance)
{
    Application app = uiDoc.Application.Application;
    Selection currentSel = uiDoc.Selection;

    PickedBox pickedBox = currentSel.PickBox (PickBoxStyle.Enclosing, "Select region of cloud for highlighting");

    XYZ min = pickedBox.Min;
    XYZ max = pickedBox.Max;

    //Transform points into filter
    View view = uiDoc.ActiveView;
    XYZ right = view.RightDirection;
    XYZ up = view.UpDirection;

    List<Plane> planes = new List<Plane>();

    // X boundaries
    bool directionCorrect = IsPointAbovePlane (right, min, max);
    planes.Add (app.Create.NewPlane (right, directionCorrect ? min : max));
    planes.Add (app.Create.NewPlane (-right, directionCorrect ? max : min));

    // Y boundaries
    directionCorrect = IsPointAbovePlane (up, min, max);
    planes.Add (app.Create.NewPlane (up, directionCorrect ? min : max));
    planes.Add (app.Create.NewPlane (-up, directionCorrect ? max : min));

    // Create filter
    PointCloudFilter filter = PointCloudFilterFactory.CreateMultiPlaneFilter (planes);
    Transaction t = new Transaction (uiDoc.Document, "Highlight");
    t.Start();
    pcInstance.SetSelectionFilter (filter);
    pcInstance.FilterAction = SelectionFilterAction.Highlight;
    t.Commit();
}
```



```

        uiDoc.RefreshActiveView();
    }

    private static bool IsPointAbovePlane ( XYZ normal, XYZ planePoint, XYZ point )
    {
        XYZ difference = point - planePoint;
        difference = difference.Normalize();
        double dotProduct = difference.DotProduct (normal);
        return dotProduct > 0;
    }

```

### 2.2.3 过滤的用户选集 ( Filtered User Selection )

PickObject()、PickObjects()和 PickElementsByRectangle()都有一个以 ISelectionFilter 作为参数的重载。ISelectionFilter 是一个接口,在选集操作期间,可用此接口实现过滤对象。它有两个可以重载的方法: AllowElement()用于指定是否允许选择某个图元, AllowReference()用于指定是否允许选择对某个几何体的参照。

代码 2-21 阐释了如何使用 ISelectionFilter 接口来限制用户对体量分类图元的选择。它不允许选择对任何几何体的参照。

**代码 2-21: 用 ISelectionFilter 来限制图元选择**

```

public static IList<Element> GetManyRefByRectangle (UIDocument doc)
{
    ReferenceArray ra = new ReferenceArray();
    ISelectionFilter selFilter = new MassSelectionFilter();
    IList<Element> eList = doc.Selection.PickElementsByRectangle (selFilter,
        "Select multiple faces") as IList<Element>;
    return eList;
}

public class MassSelectionFilter : ISelectionFilter
{
    public bool AllowElement (Element element)
    {
        if (element.Category.Name == "Mass")
        {
            return true;
        }
        return false;
    }

    public bool AllowReference (Reference refer, XYZ point)
    {
        return false;
    }
}

```

代码 2-22 演示了如何使用 ISelectionFilter 仅允许选定平表面。



代码 2-22: 用 ISelectionFilter 来限制几何体选择

```
public void SelectPlanarFaces (Autodesk.Revit.DB.Document document)
{
    UIDocument uidoc = new UIDocument (document);
    ISelectionFilter selfFilter = new PlanarFacesSelectionFilter (document);
    IList<Reference> faces = uidoc.Selection.PickObjects (ObjectType.Face,
        selfFilter, "Select multiple planar faces");
}

public class PlanarFacesSelectionFilter : ISelectionFilter
{
    Document doc = null;
    public PlanarFacesSelectionFilter (Document document)
    {
        doc = document;
    }

    public bool AllowElement (Element element)
    {
        return true;
    }

    public bool AllowReference (Reference refer, XYZ point)
    {
        if (doc.GetElement (refer).GetGeometryObjectFromReference (refer) is PlanarFace)
        {
            // Only return true for planar faces. Non-planar faces will not be selectable
            return true;
        }
        return false;
    }
}
```

从所选图元检索图元的详细信息, 请参见 1.2.4 节演练: 检索所选图元。

## 2.3 参数 (Parameters)

Revit 提供了一个通用机制, 给每个图元一组可编辑的参数。在 Revit 用户界面中, 图元属性对话框中的参数是可见的。本节介绍如何利用 Revit 平台 API 来获取和使用内置参数。关于用户定义的共享参数的更多信息, 请参阅 5.1.1 节共享参数。

在 Revit 平台 API 中, 参数在图元类中进行管理。可用以下这些方法访问参数:

- 通过遍历某图元所有参数的 `Element.Parameters` 集合 (作为例子, 请参阅 2.3.1 节演练: 获取所选图元参数)。
- 通过重载 `Element.Parameter` 属性直接访问参数。若参数不存在, 则此属性返回 `null`。
- 由 `Element.ParametersMap` 集合, 通过名称访问参数。



若已知名称字符串、内置 ID、定义或 GUID，则可从图元中检索参数对象。Parameter [String] 属性重载根据其本地化名称获取参数，如果需在多个语言区域运行，并按名称查找参数，那么编写的代码应能处理不同的语言。

Parameter [GUID] 属性重载根据其全局唯一标识 (GUID) 获取共享参数，当其创建时 GUID 即已被分配给共享参数。

### 2.3.1 演练：获取所选图元参数 (Walkthrough: Get Selected Element Parameters)

通过遍历图元参数集来检索图元参数。代码 2-23 阐释了如何从所选图元中检索参数，图 2-1 显示了获取墙参数的结果。

注：此示例中使用的一些参数成员，如 AsValueString 和 StorageType，将在本章稍后介绍。

#### 代码 2-23：获取所选图元参数

```
void GetElementParameterInformation (Document document, Element element)
{
    // Format the prompt information string
    String prompt = "Show parameters in selected Element: ";

    StringBuilder st = new StringBuilder( );
    // iterate element's parameters
    foreach (Parameter para in element.Parameters)
    {
        st.AppendLine (GetParameterInformation (para, document));
    }

    // Give the user some information
    MessageBox.Show (prompt, "Revit", MessageBoxButtons.OK);
}

String GetParameterInformation (Parameter para, Document document)
{
    string defName = para.Definition.Name + @"\t";
    // Use different method to get parameter data according to the storage type
    switch (para.StorageType)
    {
        case StorageType.Double:
            //convert the number into Metric
            defName += " : " + para.AsValueString( );
            break;
        case StorageType.ElementId:
            //find out the name of the element
            ElementId id = para.AsElementId( );
            if (id.Value >= 0)
            {
                defName += " : " + document.GetElement (ref id).Name;
            }
            else
            {

```



```
        defName += " : " + id.Value.ToString( );
    }
    break;
case StorageType.Integer:
    if ( ParameterType.YesNo == para.Definition.ParameterType )
    {
        if ( para.AsInteger( ) == 0 )
        {
            defName += " : " + "False";
        }
        else
        {
            defName += " : " + "True";
        }
    }
    else
    {
        defName += " : " + para.AsInteger( ).ToString( );
    }
    break;
case StorageType.String:
    defName += " : " + para.AsString( );
    break;
default:
    defName = "Unexposed parameter.";
    break;
}

return defName;
}
```



图 2-1 获取墙参数的结果



注：在 Revit 中，图元属性对话框的下拉列表中有某些参数值。利用 Revit 平台 API，可以获取与参数的枚举类型相一致的数值，但不能使用 `Parameter.AsValueString()` 方法来获取 `Parameter` 值的字符串。

### 2.3.2 定义 (Definition)

`Definition` 对象描述了数据类型、名称和其他参数细节，该对象派生出两种定义对象：

- `InternalDefinition` 表示全部在 Revit 数据库中的所有种类的定義。
- `ExternalDefinition` 表示存储在硬盘共享参数文件中的定义。

应该使用 `Definition` 基类来编写代码，以便代码对内部、外部参数定义都适用。代码 2-24 演示了如何使用定义类型查找特定参数。

**代码 2-24：根据定义类型查找参数**

```
//Find parameter using the Parameter's definition type.
public Parameter FindParameter (Element element)
{
    Parameter foundParameter = null;
    // This will find the first parameter that measures length
    foreach (Parameter parameter in element.Parameters)
    {
        if (parameter.Definition.ParameterType == ParameterType.Length)
        {
            foundParameter = parameter;
            break;
        }
    }
    return foundParameter;
}
```

#### 1. 参数类型 (ParameterType)

此属性返回参数的数据类型，该属性影响参数在 Revit UI 中的显示方式。`ParameterType` 枚举成员见表 2-5。

**表 2-5**

**ParameterType 枚举成员**

成员名称	说 明
Number	实数，可包括小数点
Moment	力矩
AreaForce	面力
LinearForce	线性力
Force	力
YesNo	布尔值，表示为 Yes 或 No
Material	材料属性值
URL	文本字符串，表示网址
Angle	角度。内部表示为弧度。用户可见为用户已选定的单位表示



续表

成员名称	说 明
Volume	体积。内部表示为十进制立方英尺。用户可见为用户已选择的单位表示
Area	面积。内部表示为十进制平方英尺。用户可见为用户已选择的单位表示
Integer	整数，或正或负
Invalid	参数类型无效，不应使用此值
Length	长度。内部表示为十进制英尺。用户可见为用户已选择的单位表示
Text	文本字符串

## 2. 参数组 ( ParameterGroup )

Definition 类 `ParameterGroup` 属性返回该参数定义组 ID。`BuiltInParameterGroup` 是一个 Revit 所支持的所有内置参数组枚举类型的列表。参数组用于在图元属性对话框中对参数进行分类。

## 3. 全组变化 ( VariesAcrossGroups )

此属性特定于 `InternalDefinition` 子类，对应的 `SetAllowVaryBetweenGroups()` 方法确定组实例的各相关成员此参数的值是否可以不相同。如果为 `false`，则组实例中所有相关成员此值是一致的。这只能对非内建的参数进行设置。

### 2.3.3 内建参数 ( BuiltInParameter )

Revit 平台 API 具有大量内置参数，在 `Autodesk.Revit.Parameters.BuiltInParameter` 枚举中定义（参阅 `RevitAPI Help.chm` 文件对此枚举的定义）。此枚举可以从 Visual Studio 智能感知生成如图 2-2 所示的可视文档。每个 ID 的文档包括参数名，如在英文版 Autodesk Revit 图元属性对话框中看到的一样。需要注意的是，多个不同的参数 ID 可以映射到相同的英文名称；在这些情况下，必须检查与特定图元关联的参数来确定要使用哪个参数 ID。

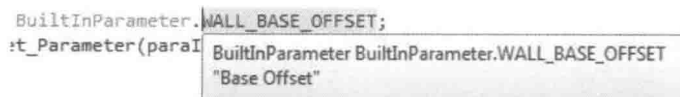


图 2-2 Visual Studio 智能感知生成的可视文档

参数 ID 用于从某个图元检索特定参数，如果它存在，则使用 `Element.Parameter` 属性检索。然而，并非所有参数都可以使用 ID 来检索。例如，族参数在 Revit 平台 API 中是未公开的，因此，无法使用内置参数 ID 来获取它们。

代码 2-25 演示了如何使用 `BuiltInParameter` ID 获取特定参数。

#### 代码 2-25：基于 BuiltInParameter 获取参数

```
public Parameter FindWithBuiltInParameterID (Wall wall)  
{  
    // Use the WALL_BASE_OFFSET paramaterId  
    // to get the base offset parameter of the wall.  
    BuiltInParameter paraIndex = BuiltInParameter.WALL_BASE_OFFSET;
```

```
Parameter parameter = wall.get_Parameter (paraIndex);

return parameter;

}
```

注意：随参数重载，可以使用枚举类型 `BuiltInParameter` 作为方法参数，如使用 `BuiltInParameter.GENERIC_WIDTH`。

如果开发人员不知道确切的 `BuiltInParameter` ID，可通过迭代 `ParameterSet` 集合来获取参数。以测试或识别为目的的另一种方法，是使用 `get_Parameter()`方法测试每个 `BuiltInParameter`。使用此方法时，有可能 `ParameterSet` 集合并未包含从 `get_Parameter` 方法返回的所有参数，尽管这很少发生。

2.3.4 存储类型 (StorageType)

`StorageType` 描述了内部存储参数值的类型。  
基于属性值，使用相应的 `Get()`和 `Set()`方法来检索和设置参数数据值。  
`StorageType` 是个枚举类型，表 2-6 列出了 Revit 支持的所有内部参数数据存储类型。

表 2-6 存 储 类 型

成员名称	说 明
String	内部数据存储为字符串
ElementId	数据类型表示一个图元并存储为一个图元 ID
Double	数据在内部存储为 8 字节浮点数
Integer	内部数据存储为有符号 32 位整数
None	代表无效的存储类型，仅内部使用

在大多数情况下，`ElementId` 值为正数，但它也可以是负数。当 `ElementId` 值为负数时，它并不表示一个图元，而是另有其意。例如，图 2-3 梁的垂直投影存储类型参数是 `ElementId`，当参数值为标高 1 或标高 2 时，`ElementId` 值为正，对应此层 `ElementId`。然而，当参数值设置为自动检测、梁中心或梁顶时，`ElementId` 值是负数。

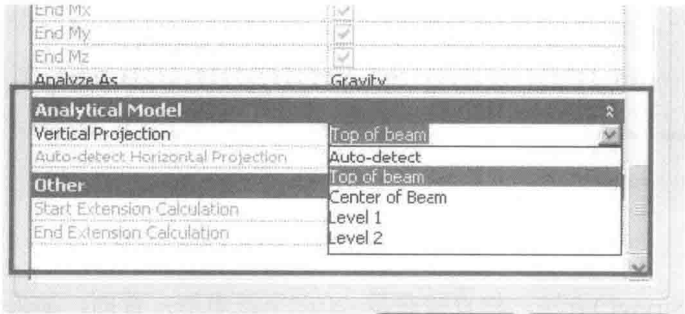


图 2-3 存储类型示例

代码 2-26 基于参数的 `StorageType`，演示了如何检查参数值是否可设置为 `double` 值。





代码 2-26: 检查参数的存储类型

```
public bool SetParameter (Parameter parameter, double value)
{
    bool result = false;
    //if the parameter is readonly, you can't change the value of it
    if (null != parameter && !parameter.IsReadOnly)
    {
        StorageType parameterType = parameter.StorageType;
        if (StorageType.Double != parameterType)
        {
            throw new Exception ("The storagetypes of value and parameter are different!");
        }

        //If successful, the result is true
        result = parameter.Set (value);
    }

    return result;
}
```

Set()方法的返回值指示参数值是否已更改。如果参数值已更改,则 Set()方法返回 true,否则返回 false。

并非所有参数都是可写的,如果对只读参数进行写入操作,则会引发异常。

### 2.3.5 AsValueString()和 SetValueString()

AsValueString()和 SetValueString()都是参数类方法。这两个方法都只适用于数值类型参数,这是一个表示测量值的双精度浮点型或整型参数。

使用 AsValueString()方法可获取带度量单位的字符串参数值。例如,基础偏移值,是一个双精度的墙参数值。在图元属性对话框中通常显示为字符串值,如-20'0",见图 2-4。

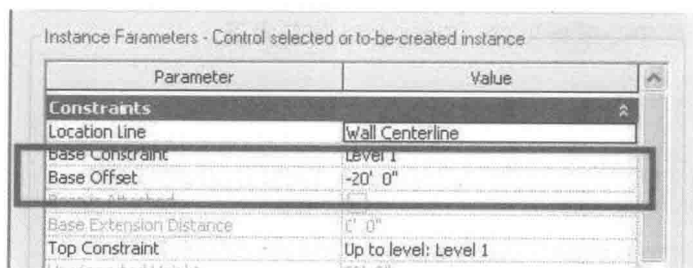


图 2-4 AsValueString 和 SetValueString 示例

使用 AsValueString()方法,将直接获得-20'0"字符串值;否则,如果使用 AsDouble()方法,则将得到没有度量单位的双精度值,即-20。

如要更改数值类型参数的值,请用 SetValueString()方法,而不是用 Set()方法。代码 2-27 阐释了如何用 SetValueString()方法更改参数值。



代码 2-27: 使用 Parameter.SetValueString() 方法

```
public bool SetWithValueString (Parameter foundParameter)
{
    bool result = false;
    if (!foundParameter.IsReadOnly)
    {
        //If successful, the result is true
        result = foundParameter.SetValueString ("-22'3'");
    }
    return result;
}
```

### 2.3.6 参数关系 (Parameter Relationships)

参数之间存在相互关系, 某个参数的值可以影响:

- 另一个参数是可设置的还是只读的。
- 图元的哪个参数是有效的。
- 另一个参数的计算值。

此外, 有些参数始终是只读的。

有些参数是 Revit 计算得出的, 如墙的长度、面积等参数。这些参数始终是只读的, 因为它们取决于图元的内部状态, 见图 2-5。

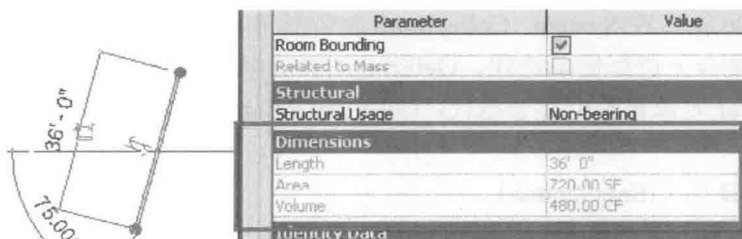


图 2-5 墙的计算参数

代码 2-28 示例中, 一个门洞的 Sill Height (门楣高度) 参数的调整, 导致 Head Height (头部高度) 参数被重新计算。

代码 2-28: 参数关系示例

```
// 洞口宜为窗洞或门洞
public void ShowParameterRelationship (FamilyInstance opening)
{
    // 获取洞口的初始 Sill Height 及 Head Height 参数
    Parameter sillPara = opening.get_Parameter (BuiltInParameter.INSTANCE_SILL_HEIGHT_PARAM);
    Parameter headPara = opening.get_Parameter (BuiltInParameter.INSTANCE_HEAD_HEIGHT_PARAM);
    double sillHeight = sillPara.AsDouble();
    double origHeadHeight = headPara.AsDouble();

    // Change the Sill Height only and notice that Head Height is recalculated
    sillPara.Set (sillHeight + 2.0);
```



```
double newHeadHeight = headPara.AsDouble();  
MessageBox.Show ("Old head height: " + origHeadHeight + "; new head height: "  
    + newHeadHeight);  
}
```

### 2.3.7 给图元添加参数 (Adding Parameters to Elements)

在 Revit 平台 API 中, 可以使用所有定义过的参数, 还可以添加用 Revit 用户界面和 Revit 平台 API 定义的自定义参数。更多有关信息, 请参阅 5.1.1 节共享参数。

## 2.4 集合 (Collections)

大多数 Revit 平台 API 的属性和方法使用 .NET 框架的集合类, 来提供对一组相关项的访问。

Revit 集合类型中实现的 `IEnumerable` 和 `IEnumerator` 接口, 在 `System.Collection` 命名空间中定义。

### 2.4.1 接口 (Interface)

以下几节讨论了与接口相关的集合类型。

#### 1. 可枚举接口 (`IEnumerable`)

`IEnumerable` 接口在 `System. Collections` 命名空间中定义。此接口公开了枚举器, 它支持在非泛型集合上进行简单迭代。`GetEnumerator()` 方法获取实现此接口的枚举器。返回的 `IEnumerator` 对象被整个集合迭代。`GetEnumerator()` 方法为 C# 中的 `foreach` 循环所隐式使用。

#### 2. 枚举器接口 (`IEnumerator`)

`IEnumerator` 接口在 `System. Collections` 命名空间中定义。它支持在非泛型集合上进行简单迭代。`IEnumerator` 是所有非泛型枚举器的基本接口。在 C# 中 `foreach` 语句掩盖了枚举器的复杂性。

注意: 建议使用 `foreach` 而不直接操控枚举器。

枚举器用于读取集合数据, 但它们不能用于修改基础集合。使用 `IEnumerator` 方法如下:

- 最初, 枚举器定位在集合中的第一个图元之前。不管怎样, 最好还是在第一次获取枚举器时, 总是调用一次 `Reset()` 方法。
  - `Reset()` 方法会将枚举器移回原始位置。在这个位置, 调用 `Current` 属性会引发异常。
  - 在读取当前迭代器值之前, 调用 `MoveNext()` 方法使枚举器前进到集合的第一个图元。
- 直到调用 `MoveNext()` 方法或 `Reset()` 方法为止, `Current` 属性总是返回同一对象。`MoveNext()` 方法使当前迭代器移到下一个图元。
- 如果 `MoveNext` 超过了集合的末尾, 则枚举器定位到集合中的最后一个图元之后, 且 `MoveNext` 返回 `false`。
  - 当枚举器在这个位置时, 随后的 `MoveNext` 调用也将返回 `false`。
  - 如果 `MoveNext` 的最后一次调用返回 `false`, 调用 `Current` 属性将引发异常。



- 若要再次设置当前迭代器到集合中的第一个图元，则调用 `Reset()` 方法，随后调用 `MoveNext()` 方法。
- 只要该集合保持不变，枚举器就仍然有效。
- 如果对集合作了更改，如添加、修改或删除图元，则枚举器失效，之后调用 `MoveNext()` 或 `Reset()` 方法将引发非法操作异常。
- 如果在 `MoveNext` 和当前迭代器之间对集合作了修改，即使枚举器已失效，`Current` 属性也会返回指定图元。

注意：所有 `Reset()` 方法调用都将导致枚举器处于同一状态。首选操作是将枚举器移到集合的开头，第一个图元之前。如果在枚举器创建之后修改了集合，则会导致枚举失效，这与 `MoveNext()` 方法和 `Current` 属性一致。

### 2.4.2 集合和迭代器 (Collections and Iterators)

在 Revit 平台 API 中，集合和迭代器是泛型和类型安全的。例如，`ElementSet` 总是包含图元，可如代码 2-29 所示使用。

**代码 2-29：使用 `ElementSet` 集合**

```
UIDocument uidoc = new UIDocument (document);
ElementSet elems = uidoc.Selection.Elements;

string info = "Selected elements: \n";
foreach (Autodesk.Revit.DB.Element elem in elems)
{
    info += elem.Name + "\n";
}

TaskDialog.Show ("Revit", info);

info = "Levels in document: \n";

FilteredElementCollector collector = new FilteredElementCollector (document);
ICollection<Element> collection = collector.OfClass (typeof (BoundaryConditions)).ToElements();
foreach (Element elem in collection)
{
    // you need not check null for elem
    info += elem.Name + "\n";
}

TaskDialog.Show ("Revit", info);
```

所有集合实现 `IEnumerable` 接口，而所有相关的迭代器实现 `IEnumerator` 接口。因此，所有方法和属性都能在 Revit 平台 API 中实现，并在相关的集合中发挥作用。

所有集合的实现都是类似的。代码 2-30 使用 `ElementSet` 和 `ModelCurveArray` 演示了如何使用集合的主要属性。

**代码 2-30: 使用集合**

```
UIDocument uidoc = new UIDocument ( document );
SelElementSet selection = uidoc.Selection.Elements;
// Store the ModelLine references
ModelCurveArray lineArray = new ModelCurveArray();

// ... Store operation
Autodesk.Revit.DB.ElementId id = new Autodesk.Revit.DB.ElementId ( 131943 ); //assume 131943 is a model line element id
lineArray.Append ( document.GetElement ( id ) as ModelLine );

// use Size property of Array
TaskDialog.Show ( "Revit", "Before Insert: " + lineArray.Size + " in lineArray." );

// use IsEmpty property of Array
if ( !lineArray.IsEmpty )
{
    // use Item (int) property of Array
    ModelCurve modelCurve = lineArray.get_Item ( 0 ) as ModelCurve;

    // erase the specific element from the set of elements
    selection.Erase ( modelCurve );

    // create a new model line and insert to array of model line
    SketchPlane sketchPlane = modelCurve.SketchPlane;

    XYZ startPoint = new XYZ ( 0, 0, 0 ); // the start point of the line
    XYZ endPoint = new XYZ ( 10, 10, 0 ); // the end point of the line
    // create geometry line
    Line geometryLine = Line.CreateBound ( startPoint, endPoint );

    // create the ModelLine
    ModelLine line =
        document.Create.NewModelCurve ( geometryLine, sketchPlane ) as ModelLine;

    lineArray.Insert ( line, lineArray.Size - 1 );
}

TaskDialog.Show ( "Revit", "After Insert: " + lineArray.Size + " in lineArray." );

// use the Clear ( ) method to remove all elements in lineArray
lineArray.Clear ( );

TaskDialog.Show ( "Revit", "After Clear: " + lineArray.Size + " in lineArray." );
```

## 2.5 编辑图元 ( Editing Elements )

通过 Revit 平台 API 可以对某个或某组图元进行移动、复制、旋转、对齐、删除、镜像、



成组和阵列等操作。API 中的编辑功能类似于 Revit 用户界面中的命令。

2.5.1 移动图元 (Moving Elements)

ElementTransformUtils 类提供了两个静态方法来将一个或多个图元从某处移到别处，见表 2-7，代码 2-31 是 MoveElement() 示例。

表 2-7 Move 方 法

成 员	说 明
MoveElement (Document, ElementId, XYZ)	根据指定向量移动文件中的一个图元
MoveElements (Document, ICollection<ElementId>, XYZ)	根据指定向量和一组 ID，移动文件中的多个图元

注意：当使用 MoveElement() 或 MoveElements() 方法时，这些方法不能将基于标高的图元移动到该标高的上方或下方。当图元基于某标高时，不能更改其 Z 坐标值，但可以将图元放置在同一标高的任何位置上。某些基于标高的图元有一个偏移实例参数可以用来在 Z 方向移动它们。例如，如果在标高 1 原点位置 (0, 0, 0) 新建一根柱，然后将其移动到新位置 (10, 20, 30)，则柱被放置在位置 (10, 20, 0)，而不是位置 (10, 20, 30)。

代码 2-31: 使用 MoveElement() 方法

```
public void MoveColumn (Autodesk.Revit.DB.Document document, FamilyInstance column)
{
    // get the column current location
    LocationPoint columnLocation = column.Location as LocationPoint;

    XYZ oldPlace = columnLocation.Point;

    // Move the column to new location.
    XYZ newPlace = new XYZ (10, 20, 30);
    ElementTransformUtils.MoveElement (document, column.Id, newPlace);

    // now get the column's new location
    columnLocation = column.Location as LocationPoint;
    XYZ newActual = columnLocation.Point;

    string info = "Original Z location: " + oldPlace.Z +
        "\nNew Z location: " + newActual.Z;

    TaskDialog.Show ("Revit", info);
}
```

- 当移动一个或多个图元时，与其相关联的图元也跟着移动。例如，如果一个有窗户的墙移动了，则窗户也会跟着移动。
- 被固定的图元 (Pinned Elements) 不能被移动。

移动 Revit 图元的另一种方法是使用 Location 及其派生对象，见代码 2-32。在 Revit 平台 API 中，Location 对象提供了图元平移和旋转的能力。使用它的派生对象，如 LocationPoint 或 LocationCurve，可以获得更多的位置信息和控制。如果 Location 图元向下



转换为 `LocationCurve` 对象或 `LocationPoint` 对象, 则可以直接将曲线或点移到新的位置上。

**代码 2-32: 使用 `Location` 进行移动**

```
bool MoveUsingLocationCurve (Autodesk.Revit.ApplicationServices.Application application, Wall wall)
{
    LocationCurve wallLine = wall.Location as LocationCurve;
    XYZ translationVec = new XYZ (10, 20, 0);
    return (wallLine.Move (translationVec));
}
```

当移动图元时, 请注意向量 (10, 20, 0) 不是目标位置, 而是偏移值。图 2-6 标示了墙在移动前和移动后的位置。

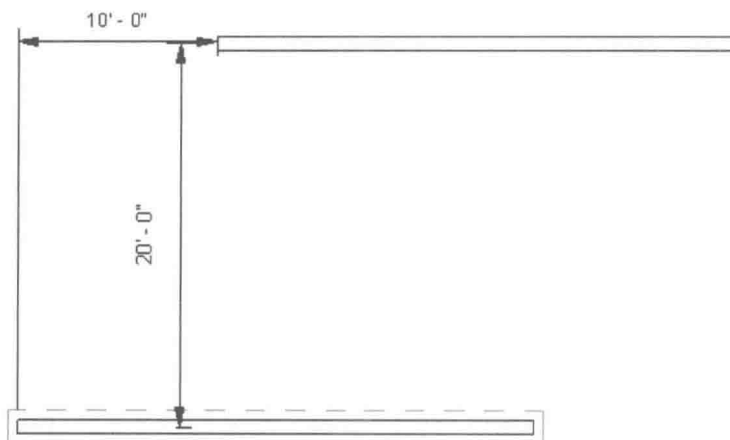


图 2-6 使用 `LocationCurve` 移动墙

此外, 您还可以使用 `LocationCurved` 的 `Curve` 属性或 `LocationPoint` 的 `Point` 属性移动某个 Revit 图元。

使用 `Curve` 属性可将曲线驱动图元移动到任何指定位置, 见代码 2-33。许多图元都是曲线驱动的, 如墙、梁和斜撑。使用此属性还可以调整图元的长度。

**代码 2-33: 使用 `Curve` 进行移动**

```
void MoveUsingCurveParam (Autodesk.Revit.ApplicationServices.Application application, Wall wall)
{
    LocationCurve wallLine = wall.Location as LocationCurve;
    XYZ p1 = XYZ.Zero;
    XYZ p2 = new XYZ (10, 20, 0);
    Line newWallLine = Line.CreateBound (p1, p2);

    // Change the wall line to a new line.
    wallLine.Curve = newWallLine;
}
```

使用 `LocationCurve.JoinType` 属性, 可以获取或设置基于曲线图元的连接属性, 使用 `LocationPoint` 的 `Point` 属性可设置图元的物理位置, 见代码 2-34。



代码 2-34: 使用 Point 进行移动

```
void LocationMove (FamilyInstance column)
{
    LocationPoint columnPoint = column.Location as LocationPoint;
    if (null != columnPoint)
    {
        XYZ newLocation = new XYZ (10, 20, 0);
        // Move the column to the new location
        columnPoint.Point = newLocation;
    }
}
```

## 2.5.2 复制图元 (Copying Elements)

ElementTransformUtils 类提供了几个静态方法, 从某处复制一个或多个图元到别处, 见表 2-8, 可以复制到同一文件或视图内, 也可以复制到其他文件或视图内。

表 2-8 Copy 方 法

成 员	说 明
CopyElement (Document, ElementId, XYZ)	复制图元并将其置于给定转换所指定的位置
CopyElements (Document, ICollection<ElementId>, XYZ)	复制一组图元并将其置于给定转换所指定的位置
CopyElements (Document, ICollection<ElementId>, Document, Transform, CopyPasteOptions)	从源文件复制一组图元到目标文件
CopyElements (View, ICollection<ElementId>, View, Transform, CopyPasteOptions)	从源视图复制一组图元到目标视图

所有这些方法都会返回新建图元的 ElementIds 集合, 包括 CopyElement()方法。集合包含因依赖关系所创建的任何图元。

从一个文件复制到另一文件的方法仅可用于复制非视图专有图元。副本放置在各自原始位置或由可选转换所指定的位置。

视图专有图元应使用从一个视图复制到另一个视图的复制方法。该方法既可用于视图专有图元又可用于模型图元。然而, 对于模型图元, 绘图视图不能作为目标视图。被粘贴的图元被重新定位, 以确保放置在目标视图的正确位置上。例如, 当从某个标高复制到另一标高时, 高程被改变了。通过提供可选转换参数, 在目标视图中还可执行一个附加转换。此附加转换必须在目标视图平面内。

从一个视图复制到另一个视图时, 源视图和目标视图都必须是能够绘制详图和视图专有图元的二维图形视图, 如楼板和天花板平面、立面、剖面或绘图视图。当从源视图复制到目标视图时, ElementTransformUtils.GetTransformFromViewToView()方法将返回应用于图元的转换。

在视图之间或文件之间复制时, 可以设置一个可选的 CopyPasteOptions 参数来覆盖默认 copy/paste 设置。默认情况下, 在粘贴操作期间重复的类型名称事件中, Revit 显示一个模态对话框选项: 只复制有唯一名称的类型, 或取消操作。CopyPasteOptions 可用于指定一个自定义处理程序, 使用 IDuplicateTypeNamesHandler 接口, 处理重复的类型名称。

有关在视图之间或文件之间复制的详例, 请参阅 Revit SDK 中的 Duplicate Views 样例。





### 2.5.3 旋转图元 (Rotating Elements)

ElementTransformUtils 类提供了两种静态方法, 用于旋转项目中的一个或多个图元, 见表 2-9 和代码 2-35。

表 2-9 Rotate 方 法

成 员	说 明
RotateElement (Document, ElementId, Line, double)	通过指定绕给定轴的弧度数, 旋转文档中的单个图元
RotateElements (Document, ICollection<ElementId>, Line, double)	根据项目中的图元 IDs 并指定绕给定轴的弧度数, 旋转文档中的多个图元

在这些方法中, 旋转角度是以弧度表示的。正的弧度表示绕指定轴逆时针旋转, 负的弧度表示顺时针旋转, 见图 2-7 和图 2-8。

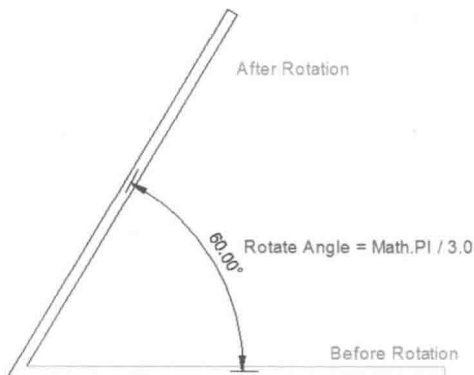


图 2-7 逆时针旋转

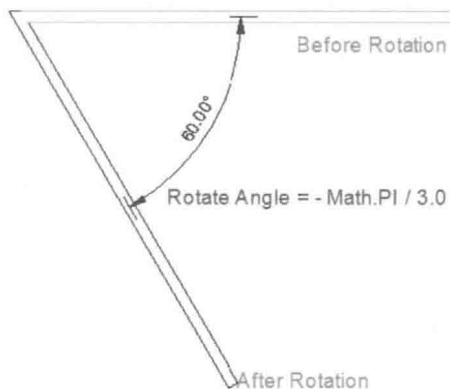


图 2-8 顺时针旋转

注意: 被锁定图元不能旋转。

#### 代码 2-35: 使用 RotateElement()方法

```
public void RotateColumn (Autodesk.Revit.DB.Document document, Autodesk.Revit.DB.Element element)
{
    XYZ point1 = new XYZ (10, 20, 0);
    XYZ point2 = new XYZ (10, 20, 30);
    // The axis should be a bound line.
    Line axis = Line.CreateBound (point1, point2);
    ElementTransformUtils.RotateElement (document, element.Id, axis, Math.PI / 3.0);
}
```

如果图元的 Location 可以向下转换为 LocationCurve 或 LocationPoint, 就可以直接旋转该曲线或点, 见代码 2-36 和代码 2-37。

#### 代码 2-36: 基于定位曲线旋转

```
bool LocationRotate (Autodesk.Revit.ApplicationServices.Application application, Autodesk.Revit.DB.Element element)
{
    bool rotated = false;
```



```
// Rotate the element via its location curve.
LocationCurve curve = element.Location as LocationCurve;
if (null != curve)
{
    Curve line = curve.Curve;
    XYZ aa = line.GetEndPoint (0);
    XYZ cc = new XYZ (aa.X, aa.Y, aa.Z + 10);
    Line axis = Line.CreateBound (aa, cc);
    rotated = curve.Rotate (axis, Math.PI / 2.0);
}

return rotated;
}
```

#### 代码 2-37: 基于定位点旋转

```
bool LocationRotate (Autodesk.Revit.ApplicationServices.Application application, Autodesk.Revit.Element element)
{
    bool rotated = false;
    LocationPoint location = element.Location as LocationPoint;

    if (null != location)
    {
        XYZ aa = location.Point;
        XYZ cc = new XYZ (aa.X, aa.Y, aa.Z + 10);
        Line axis = Line.CreateBound (aa, cc);
        rotated = location.Rotate (axis, Math.PI / 2.0);
    }

    return rotated;
}
```

### 2.5.4 对齐图元 (Aligning Elements)

ItemFactoryBase.NewAlignment()方法可以在两个参照之间新建一个锁定对齐。这两个参照必须为下列组合之一:

- 两个平表面。
- 两条线。
- 直线和点。
- 直线和参照平面。
- 2 条弧。
- 2 个圆柱面。

这些参照必须已经是几何对齐的, 因为该函数不会强制它们变成对齐。如果对齐可被创建, 则将返回一个表示锁定对齐的新 Dimension 对象; 否则将引发异常。

NewAlignment()方法还需要一个视图, 用来确定对齐的方向。



请参阅包含在 SDK Samples FamilyCreation 文件夹中的 CreateTruss 示例。它有几个使用 NewAlignment() 的例子，如锁定新桁架下弦到底部参照平面。

## 2.5.5 镜像图元 ( Mirroring Elements )

ElementTransformUtils 类提供了两个静态方法来镜像项目中的一个或多个图元，见表 2-10。

表 2-10 Mirror 方 法

成 员	说 明
MirrorElement ( Document, ElementId, Plane )	以一个几何平面镜像单个图元
MirrorElements ( Document, ICollection<ElementId>, Plane )	以一个几何平面镜像多个图元

在执行镜像操作之后，即可从 Selection ElementSet 访问新图元。

ElementTransformUtils.CanMirrorElement() 和 ElementTransformUtils.CanMirrorElements() 可用于在尝试镜像一个图元之前，确定一个或多个图元是否可以镜像。

代码 2-38 说明了如何使用基于墙侧面计算出的平面来镜像一面墙。

### 代码 2-38：镜像一面墙

```
public void MirrorWall ( Autodesk.Revit.DB.Document document, Wall wall )
{
    Reference reference = HostObjectUtils.GetSideFaces ( wall, ShellLayerType.Exterior ).First();

    // get one of the wall's major side faces
    Face face = wall.GetGeometryObjectFromReference ( reference ) as Face;

    UV bboxMin = face.GetBoundingBox().Min;
    // create a plane based on this side face with an offset of 10 in the X & Y directions

    Plane plane = new Plane ( face.ComputeNormal ( bboxMin ),
                             face.Evaluate ( bboxMin ).Add ( new XYZ ( 10, 10, 0 ) ) );

    ElementTransformUtils.MirrorElement ( document, wall.Id, plane );
}
```

每个族实例都具有 Mirrored 属性，它显示该族实例（如柱）是否已被镜像。

## 2.5.6 成组图元 ( Grouping Elements )

Revit 平台 API 用 Creation. Document. NewGroup() 方法选择一个图元或多个图元与组，然后将它们合并，见代码 2-39。对于某个组的每个实例，它们之间是有关联的。例如，项目中创建包含一张床、墙壁和窗户的多个实例，如果修改该组中的一面墙，那么该组中的所有实例该类型墙都会改变。由于一次操作就可更改某个组的多个实例，因此大大简化了建筑模型的修改。

**代码 2-39: 创建一个组**

```
Group group = null;
UIDocument uidoc = new UIDocument (document);
ElementSet selection = uidoc.Selection.Elements;
if (selection.Size > 0)
{
    // Group all selected elements
    group = document.Create.NewGroup (uidoc.Selection.GetElementIds());
}
```

最初, 该组有一个类属名称, 如 Group1。但可通过更改组名来修改它, 见代码 2-40。

**代码 2-40: 给组重命名**

```
// Change the default group name to a new name "MyGroup"
group.GroupType.Name = "MyGroup";
```

Revit 有三种类型的组: 模型组、详图组和附属详图组。所有这些组都使用 NewGroup() 方法来创建。所创建组的类型取决于组内图元的类型。

- 若无任何详图图元, 则创建的是个模型组。
- 如果所有的图元都是详图图元, 那么创建的是个详图组。
- 如果这两种类型的图元都包括在内, 则创建一个包含附属详图组的模型组并返回。

注意: 图元成组后, 可将图元从项目中删除。

- 模型组中某个模型图元被删除时, 即使应用程序成功返回到用户界面, 但若单击该组或鼠标悬停于其上, 则它仍然可见。事实上, 该模型图元已被删除, 并且无法选择或访问该图元。
- 当项目中一组实例的最后一个成员被删除、剔除或移除时, 模型组实例也就被删除了。

图元成组后, 就不能移动或旋转。如果在已成组的图元上执行这些操作, 尽管 Move() 或 Rotate() 方法返回 true, 但实际上图元没有发生任何变化。

如果参照的图元没有成组, 则无法将其尺寸和标记成组。如果这样做则 API 调用会失败。

开发人员可以对参照某个模型组中模型图元的那些尺寸和标记进行分组。这些尺寸和标记被添加到一个附属详图组。你无法单独移动、复制、旋转、阵列或镜像附属详图组, 而不对其参照模型组执行相同操作。

### 2.5.7 创建图元阵列 (Creating Arrays of Elements)

Revit 平台 API 提供了两个类, 即 LinearArray 和 RadialArray 来阵列项目中的一个或多个图元。这些类提供静态方法来创建一个或多个选定构件的线性或径向阵列。线性阵列表示从一个点顺着一条线创建的阵列, 而径向阵列表示沿弧线创建的阵列。

可以选择同一面墙上的一扇门和一扇窗, 然后通过阵列创建门、墙、窗结构布局的多个实例。

LinearArray 和 RadialArray 还提供了一些方法, 来阵列未被分组和关联的一个或多个



图元, 见表 2-11。虽然类似于阵列图元的 `Create()` 方法, 但每个目标图元都独立于其他图元, 可对其操作而不影响其他图元。有关更多信息, 请参阅表 2-11 和表 2-12 所列创建线性或径向阵列的可用方法。

**表 2-11** **LinearArray 方法**

成 员	说 明
<code>Create(Document, View, ElementId, int, XYZ, ArrayAnchorMember)</code>	根据指定数目阵列项目中的单个图元
<code>Create(Document, View, ICollection&lt;ElementId&gt;, int, XYZ, ArrayAnchorMember)</code>	根据指定数目阵列项目中的一组图元
<code>ArrayElementWithoutAssociation(Document, View, ElementId, int, XYZ, ArrayAnchorMember)</code>	根据指定数目阵列项目中的单个图元, 目标图元与线性阵列不关联
<code>ArrayElementsWithoutAssociation(Document, View, ICollection&lt;ElementId&gt;, int, XYZ, ArrayAnchorMember)</code>	根据指定数目阵列项目中的一组图元, 目标图元与线性阵列不关联

**表 2-12** **RadialArray 方法**

成 员	说 明
<code>Create(Document, View, ElementId, int, Line, double, ArrayAnchorMember)</code>	根据输入的旋转轴, 阵列项目中的单个图元
<code>Create(Document, View, ICollection&lt;ElementId&gt;, int, Line, double, ArrayAnchorMember)</code>	根据输入的旋转轴, 阵列项目中的一组图元
<code>ArrayElementWithoutAssociation(Document, View, ElementId, int, Line, double, ArrayAnchorMember)</code>	根据输入的旋转轴, 阵列项目中的单个图元, 目标图元与径向阵列不关联
<code>ArrayElementsWithoutAssociation(Document, View, ICollection&lt;ElementId&gt;, int, Line, double, ArrayAnchorMember)</code>	根据输入的旋转轴, 阵列项目中的一组图元, 目标图元与径向阵列不关联

如果需要创建一个构件的多个实例并同时操控它们, 则可使用阵列图元的方法。阵列中的每个实例都可以是组的成员。

注意: 使用阵列图元方法, 适用以下规则:

- (1) 在执行线性 and 径向阵列操作时, 依赖于这些阵列图元的图元也会被阵列。
- (2) 不能成组的那些图元无法阵列。更多信息请参阅 Revit 用户指南中关于组和阵列的限制条件的内容。
- (3) 大部分注释符号不支持阵列。

### 2.5.8 删除图元 (Deleting Elements)

Revit 平台 API 提供了 `Delete()` 方法来删除项目中的一个或多个图元, 见表 2-12。

**表 2-13** **删 除 成 员**

成 员	说 明
<code>Delete(ElementId)</code>	使用图元 ID 删除项目中的单个图元
<code>Delete(ICollection&lt;ElementId&gt;)</code>	使用图元 IDs 删除项目中的多个图元

根据图元 ID 删除单个图元, 见代码 2-41。

**代码 2-41: 根据图元 ID 删除单个图元**

```
private void DeleteElement (Autodesk.Revit.DB.Document document, Element element)
{
    // Delete an element via its id
    Autodesk.Revit.DB.ElementId elementId = element.Id;
    ICollection<Autodesk.Revit.DB.ElementId> deletedIdSet = document.Delete (elementId);

    if (0 == deletedIdSet.Count)
    {
        throw new Exception ("Deleting the selected element in Revit failed.");
    }

    String prompt = "The selected element has been removed and ";
    prompt += deletedIdSet.Count - 1;
    prompt += " more dependent elements have also been removed.";

    // Give the user some information
    TaskDialog.Show ("Revit", prompt);
}
```

注意: 删除图元时, 与其关联的所有子图元也会被删除, 如代码 2-41 所示。  
API 还提供了删除多个图元的方法, 见代码 2-42。

**代码 2-42: 根据图元 ID 删除多个图元**

```
// Delete all the selected elements via the set of element ids.
ICollection<Autodesk.Revit.DB.ElementId> idSelection = null ;
UIDocument uidoc = new UIDocument (document);
foreach (Autodesk.Revit.DB.Element elem in uidoc.Selection.Elements)
{
    Autodesk.Revit.DB.ElementId id = elem.Id;
    idSelection.Add (id);
}

ICollection<Autodesk.Revit.DB.ElementId> deletedIdSet = document.Delete (idSelection);

if (0 == deletedIdSet.Count)
{
    throw new Exception ("Deleting the selected elements in Revit failed.");
}

TaskDialog.Show ("Revit", "The selected element has been removed.");
```

注意: 删除图元后, 删除图元的参照会失效, 如果访问它们将引发异常。

**2.5.9 锁定图元 (Pinned Elements)**

图元可以锁定, 以防其移动。Element.Pinned 属性可用于检查某个图元是否已锁定或



未锁定于另一图元。

当 `Element.Pinned` 设置为 `true`，则不能移动或旋转该图元。

## 2.6 视图 (Views)

视图是从 Revit 模型产生的图像，其对文件中存储的数据有特许访问权。它们可以是图形，如平面图；或文本，如明细表。每个项目文件都有一个或多个不同视图。最后一个焦点窗口是活动视图。

`Autodesk.Revit.DB.View` 类是 Revit 文件中所有视图类型的基类。`Autodesk.Revit.UI.UIView` 类表示在 Revit 用户界面中的窗口视图。

### 2.6.1 关于视图 (About Views)

#### 1. 视图过程 (View Process)

图 2-9 说明了如何生成一个视图。

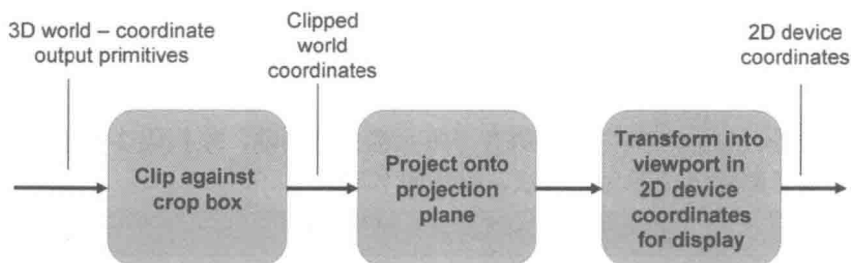


图 2-9 创建视图的过程

每个视图都是由三维对象投影到二维投影面而生成的。投影分为两个基本类：

- Perspective (透视投影)。
- Parallel (平行投影)。

在投影类型确定之后，必须指定所需三维模型和场景渲染的一些条件。关于投影的更多信息，请参阅 2.6.2 节 View 3D。

全局坐标包括以下内容：

- 观察者眼睛的位置。
- 投影显示的视图平面位置。

Revit 使用以下两个坐标系：

- 建筑所在的全局或模型空间坐标系。
- 视图坐标系。

视图坐标系表示模型在观察者视野里如何显示。其原点是观察者的眼睛位置，在模型空间中的坐标可通过 `View.Origin` 属性来检索。`X`、`Y` 和 `Z` 轴则分别由 `View.RightDirection`、`View.UpDirection` 和 `View.ViewDirection` 属性表示。

- `View.RightDirection`：朝向屏幕右方。
- `View.UpDirection`：朝向屏幕上方。



- View.ViewDirection: 从屏幕指向观察者。

视图坐标系符合右手螺旋规则。有关更多信息, 请参阅 2.6.2 节 View 3D 中透视投影图及平行投影图。

三维模型空间中不显示的某些部分, 如那些在观察者身后或是因太远而不能清晰显示的部分, 这在投影到投影平面之前就已被排除。此操作需要裁剪视图。裁剪适用规则如下:

- (1) 裁剪区域之外的图元不再出现在视图中。
- (2) View.GetCropRegionShapeManager 方法返回一个 ViewCropRegionShapeManager, 提供裁剪区域的边界信息, 裁剪区域可能是矩形也可能不是。
- (3) View.CropBoxVisible 属性确定在视图中裁剪框是否可见。
- (4) View.CropBoxActive 属性确定裁剪框是否确实被用于裁剪视图。

裁剪后, 模型投射到投影面。投影适用规则如下:

- (1) 投影内容映射到屏幕视图端口以显示。
- (2) 在映射过程中, 会缩放投影内容, 以便它们能正确显示在屏幕上。
- (3) View.Scale 属性是实际模型尺寸与视图尺寸的比例。
- (4) 图纸上的视图边界是裁剪区域, 这是裁剪形状在投影平面上的投影。
- (5) 裁剪区域的大小和位置由 View.

OutLine 属性所决定。

## 2. 显示设置 (Display Settings)

视图类属性可获取和设定显示样式设置和详细程度设置。View.DisplayStyle 属性使用 DisplayStyle 枚举和 Revit 窗口底部对应的显示可用选项, 见图 2-10。

由于设置视图的显示样式为光线追踪会进入特定的性能有限的受限模式, 因此不允许对此值直接指定显示样式。

View.DetailLevel 属性使用 ViewDetailLevel 枚举和 Revit 窗口底部对应的详细程度可用选项, 见图 2-11。

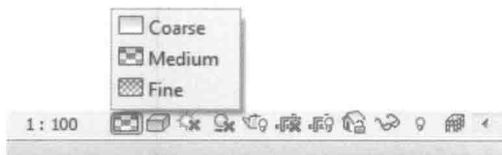


图 2-11 Revit 窗口底部对应的详细程度可用选项



图 2-10 Revit 窗口底部对应的显示可用选项

ViewDetailLevel 枚举亦包含给定视图不使用详细程度的未定义情况。

## 3. 临时视图模式 (Temporary View Modes)

视图类允许开发人员控制临时视图模式。

View.EnableRevealHiddenMode() 方法对视图中的隐藏图元启用显示模式。View.EnableTemporaryAnalyticalDisplayMode() 方法仅对分析模型类别的图元启用临时显示模式。而 View.DisableTemporaryViewMode() 方法则禁用指定的临时视图模式。DisableTemporaryViewMode() 方法采用 TemporaryViewMode 枚举, 可能的选项见表 2-14。





表 2-14 DisableTemporaryViewMode() 方法选项

成 员 名 称	说 明
RevealHiddenElements	显示隐藏图元模式
TemporaryHideIsolate	临时隐藏/隔离模式
WorksharingDisplay	工作组显示模式之一
AnalyticalModel	隔离分析模型模式
Rayrace	交互式光线追踪模型显示模式

View.IsInTemporaryViewMode() 方法可用于确定当前视图是否处于指定的临时视图模式。

#### 4. 视图中图元可见性 ( Element Visibility in a View )

视图跟踪可见图元。可以用一个文件和视图 ID 构建的 FilteredElementCollector 来检索该视图中可见的图形图元。该集合中可能会有某些隐藏的或被其他图元覆盖了图元，可以通过旋转视图或移除覆盖图元查看它们。访问这些可见图元可能需要 Revit 重建视图的几何图形。当程序代码第一次对某给定视图使用此构造函数，或第一次对显示设置刚被更改的视图使用此构造函数时，性能可能存在明显的下降。

图元在视图中显示还是隐藏，按其类别确定。

- View.GetVisibility() 方法查询类别以确定它是否在视图中可见。
- View.SetVisibility() 方法将特定类别的所有图元设置为可见或不可见。

基于视图的 FilteredElementCollector 只包含当前视图中的可见图元，无法检索到非图形图元或不可见图元。而基于文件的 FilteredElementCollector 检索文件中所有的图元，包括不可见图元和非图形图元。例如，当在一个空项目中创建一个默认的三维视图时，视图没有任何图元，但文件中有很多图元，所有这些图元都是不可见图元。

代码 2-43 计算了活动文件中和活动视图中墙类别图元的数量。活动视图中的图元数量不同于文件中的图元数量，因为文件中还包含非图形的墙类别图元。

代码 2-43: 计算活动视图中的图元

```
private void CountElements ( Autodesk.Revit.DB.Document document )
{
    StringBuilder message = new StringBuilder( );
    FilteredElementCollector viewCollector = new
        FilteredElementCollector ( document, document.ActiveView.Id );
    viewCollector.OfCategory ( BuiltInCategory.OST_Walls );
    message.AppendLine ( "Wall category elements within active View: "
        + viewCollector.ToElementIds( ).Count );

    FilteredElementCollector docCollector = new FilteredElementCollector ( document );
    docCollector.OfCategory ( BuiltInCategory.OST_Walls );
    message.AppendLine ( "Wall category elements within document: "
        + docCollector.ToElementIds( ).Count );

    TaskDialog.Show ( "Revit", message.ToString( ) );
}
```

临时视图模式会影响图元的可见性。View.IsInTemporaryViewMode()方法可用于确定视图是否处于某个临时视图模式。View.IsElementVisibleInTemporary ViewMode()方法识别在指定视图模式下某个图元是否可见。这仅适用于 TemporaryHideIsolate 和 AnalyticalModel 视图模式，其他模式将导致异常。

5. 创建和删除视图 (Creating and Deleting Views)

Revit 平台 API 提供了许多方法，来创建由 Autodesk.Revit. DB. View 派生的相应视图图元。大多数视图类型是使用派生出的视图类静态方法创建的。如果成功地创建了一个视图，则这些方法会返回一个对视图的引用，否则会返回 null。以下各节将介绍具体到每个视图类的方法。

还可以用 View.Duplicate()创建视图。可以从一个现有视图，带有新视图细节或依赖的选项，来创建一个新视图。

使用 Document.Delete()方法可以删除带有 ID 的视图，还可删除与视图关联的图元。例如，删除标高图元会导致 Revit 删除相应的平面视图，而删除摄像图元会导致 Revit 删除相应的三维视图。

2.6.2 视图类型 (View Types)

不同类型的 Revit 视图由 Revit API 中不同的类表示。有关各种视图类型的详细信息，请参阅下述专题。

1. 概述 (Overview)

项目模型可以有多个视图类型。图 2-12 显示了项目浏览器中不同类型的视图。

在 API 中，有三种方法可用于分类视图。第一种方法是使用视图图元 View.ViewType 属性。它返回一个指示视图类型的枚举值。表 2-15 列出了所有可用的视图类型。

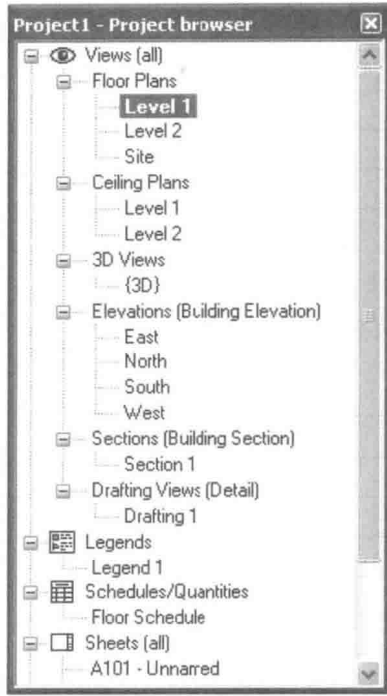


图 2-12 项目浏览器中的不同视图

表 2-15 视图类型

成员名	说明	成员名	说明
AreaPlan	面积视图	Legend	图例视图
CeilingPlan	天花板投影平面视图	LoadsReport	负荷报表视图
ColumnSchedule	柱明细表视图	PanelSchedule	配电盘视图
CostReport	成本报表视图	PressureLossReport	压降报表视图
Detail	详图视图	Rendering	渲染视图
DraftingView	绘图视图	Report	报表视图
DrawingSheet	绘图图纸视图	Schedule	明细表视图
Elevation	立面视图	Section	横剖面视图
EngineeringPlan	工程视图	ThreeD	三维视图
FloorPlan	楼板平面视图	Undefined	未定义/未指定视图
Internal	Revit 内部视图	Walkthrough	漫游视图



第二种方法是按类的类型进行视图分类。表 2-16 列出了在项目浏览器中的视图类型和相应视图。

表 2-16 视图类型和相应视图

项目浏览器视图	视图类型	类的类型
Area Plans	ViewType.AreaPlan	Elements.ViewPlan
Ceiling Plans	ViewType.CeilingPlan	Elements.ViewPlan
Graphic Column Schedule	ViewType.ColumnSchedule	Elements.View
Detail Views	ViewType.Detail	Elements.ViewSection
Drafting Views	ViewType.DraftingView	Elements.ViewDrafting
Sheets	ViewType.DrawingSheet	Elements.ViewSheet
Elevations	ViewType.Elevation	Elements.ViewSection
Structural Plans (Revit Structure)	ViewType.EngineeringPlan	Elements.ViewPlan
Floor Plans	ViewType.FloorPlan	Elements.ViewPlan
Legends	ViewType.Legend	Elements.View
Reports (Revit MEP)	ViewType.LoadsReport	Elements.View
Reports (Revit MEP)	ViewType.PanelSchedule	Elements.PanelScheduleView
Reports (Revit MEP)	ViewType.PresureLossReport	Elements.View
Renderings	ViewType.Rendering	Elements.ViewDrafting
Reports	ViewType.Report	Elements.View
Schedules/Quantities	ViewType.Schedule	Elements.ViewSchedule
Sections	ViewType.Section	Elements.ViewSection
3D Views	ViewType.ThreeD	Elements.View3D
Walkthroughs	ViewType.Walkthrough	Elements.View3D

代码 2-44 演示了如何使用视图的 ViewType 属性，以确定视图的类型。

代码 2-44：确定视图类型

```
public void GetViewType (Autodesk.Revit.DB.View view)
{
    // Get the view type of the given view, and format the prompt string
    String prompt = "The view is ";

    switch (view.ViewType)
    {
        case ViewType.AreaPlan:
            prompt += "an area view.";
            break;
        case ViewType.CeilingPlan:
            prompt += "a reflected ceiling plan view.";
            break;
        case ViewType.ColumnSchedule:
            prompt += "a column schedule view.";
            break;
```



```

case ViewType.CostReport:
    prompt += "a cost report view.";
    break;
case ViewType.Detail:
    prompt += "a detail view.";
    break;
case ViewType.DraftingView:
    prompt += "a drafting view.";
    break;
case ViewType.DrawingSheet:
    prompt += "a drawing sheet view.";
    break;
case ViewType.Elevation:
    prompt += "an elevation view.";
    break;
case ViewType.EngineeringPlan:
    prompt += "an engineering view.";
    break;
case ViewType.FloorPlan:
    prompt += "a floor plan view.";
    break;
// ...
default:
    break;
}
// Give the user some information
MessageBox.Show (prompt, "Revit", MessageBoxButtons.OK);
}

```

第三种对视图进行分类的方法是使用 `ViewFamilyType` 类。大多数视图创建方法需要新视图 `ViewFamilyType` 的 ID。`ViewFamilyType` 的 ID 可以由 `View.GetTypeId()` 方法派生。`ViewFamilyType.ViewFamily` 属性返回一个 `ViewFamily` 枚举，以指定 `ViewFamilyType` 族，类似代码 2-44 记录的 `ViewType` 枚举。代码 2-45 演示了如何从视图获取 `ViewFamily`。

#### 代码 2-45：从 `ViewFamilyType` 确定视图类型

```

public ViewFamily GetViewFamily (Document doc, View view)
{
    ViewFamily viewFamily = ViewFamily.Invalid;

    ElementId viewTypeId = view.GetTypeId();
    if (viewTypeId.IntegerValue > 1) // some views may not have a ViewFamilyType
    {
        ViewFamilyType viewFamilyType = doc.GetElement (viewTypeId) as ViewFamilyType;
        viewFamily = viewFamilyType.ViewFamily;
    }

    return viewFamily;
}

```



## 2. 三维视图 (View3D)

View3D 属于自由方向三维视图。三维视图有两种, 即透视视图和正等轴测视图, 在 Revit 用户界面中也称为正交投影视图。两种视图基于投影射线关系而不同。View3D.IsPerspective 属性可指明三维视图是透视视图还是正等轴测视图。

(1) 透视视图 (Perspective View)。图 2-13 说明了如何创建透视视图。

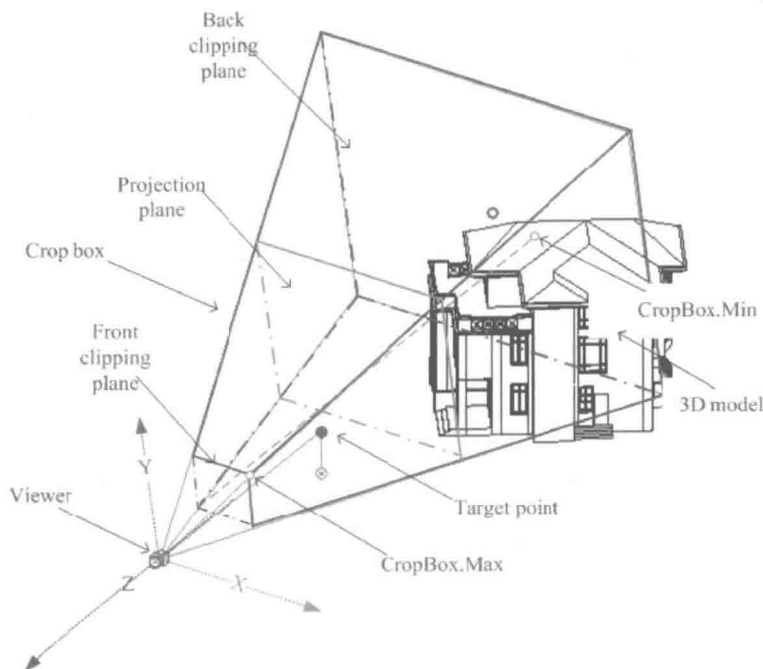


图 2-13 透视投影

- 直接投影光线穿过模型中的每个点与投影面相交形成投影内容。
  - 为了方便从全局坐标转换到视图平面, 视图坐标系是基于观察者所处位置的。
  - 其原点由 View.Origin 属性表示, 是观察者所处的位置。
  - 观察者的全局坐标用 ViewOrientation3D.EyePosition() 属性 [ 由 View3D.EyePosition() 检索 ] 检索。因此, 三维视图中, View.Origin 总是对等于相应的 ViewOrientation3D.EyePosition。
  - 如前所述, 视图坐标系的确定如下:
    - X 轴由 View.RightDirection 确定。
    - Y 轴由 View.UpDirection 确定。
    - Z 轴由 View.ViewDirection 确定。
  - 三维空间中, 视图方向是从目标点指向观察者, 屏幕空间中, 是从屏幕指向观察者。
- 静态方法 View3D.CreatePerspective() 可用于创建新的透视视图, 见代码 2-46。

### 代码 2-46: View3D.CreatePerspective() 方法

```
public static View3D View3D.CreatePerspective (Document document, ElementId viewFamilyTypeId;
```



ViewFamilyTypeId 参数需要是三维的 ViewType。

代码 2-47 阐释了如何创建透视三维视图。

代码 2-47: 创建一个透视画法的三维视图

```
// Find a 3D view type
IEnumerable<ViewFamilyType> viewFamilyTypes = from elem in new
FilteredElementCollector (document) .OfClass (typeof (ViewFamilyType))
    let type = elem as ViewFamilyType
    where type.ViewFamily == ViewFamily.ThreeDimensional
    select type;

// Create a new Perspective View3D
View3D view3D = View3D.CreatePerspective (document, viewFamilyTypes.First().Id);
if (null != view3D)
{
    // By default, the 3D view uses a default orientation.
    // Change the orientation by creating and setting a ViewOrientation3D
    XYZ eye = new XYZ (0, -100, 10);
    XYZ up = new XYZ (0, 0, 1);
    XYZ forward = new XYZ (0, 1, 0);
    view3D.SetOrientation (new ViewOrientation3D (eye, up, forward));

    // turn off the far clip plane with standard parameter API

    Parameter farClip = view3D.get_Parameter ("Far Clip Active");
    farClip.Set (0);
}
```

透视裁剪框是以观察者的位置为顶点的棱锥的一部分。它是两个平行的剪辑平面之间的几何体。裁剪框限定模型被剪辑的部分，并将其投射到视图平面。

- 裁剪框由 View.CropBox 属性表示，返回一个 BoundingBoxXYZ 对象。
- CropBox.Min 和 CropBox.Max 两个点标记在前文图上 (图 2-13)。注意，透视视图中的 CropBox.Min 点，是将裁剪框前剪辑平面投影到后剪辑平面而生成。

裁剪框坐标是基于视图坐标系的坐标。使用 Transform.OfPoint() 转换 CropBox.Min 和 CropBox.Max 到全局坐标系。关于转换的更多细节，请参阅 3.7.3 中 Geometry.Transform。

投影平面和前、后剪辑平面都垂直于视图方向。CropBox.Max 和 CropBox.Min 两点之间连线平行于视图方向。有了这些要素，就可以计算出裁剪框的几何形状。

裁剪区域是投影面和裁剪框的矩形交集，见图 2-14。

- 使用 View.CropRegion 属性检索几何信息。此属性返回一个 BoundingBoxUV 实例。
- View.OutLine.Max 属性表示右上角。
- View.OutLine.Min 属性表示左下角。
- 如同裁剪框一样，裁剪区域的坐标也基于视图坐标系。以下表达式是个等式。

$$\text{View.CropBox.Max.X (Y)} / \text{View.OutLine.Max.X (Y)} = \text{View.CropBox.Min.X (Y)} / \text{View.OutLine.Min.X (Y)}$$

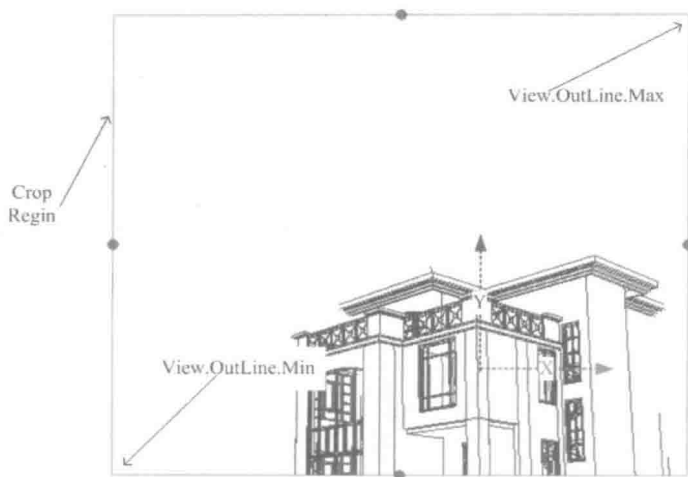


图 2-14 三维透视视图

由于对象的透视投影的大小与对象到投影中心的距离成反比, 因此比例对于透视视图是没有意义的。三维透视视图 `Scale` 属性总是返回零值。

(2) 正等轴测视图 (Isometric View)。静态方法 `View3D.CreateIsometric()`, 可以创建一个新的正等轴测视图, 见代码 2-48。

用平行投影光线将模型投射到与光线正交的平面上, 生成正等轴测视图, 见图 2-15。视图坐标系类似于透视视图坐标系统, 但裁剪框是一个平行六面体, 其各个面分别与投影光线平行或垂直。`View.CropBox` 属性表示两个对角, 其坐标亦是基于视图坐标系的。

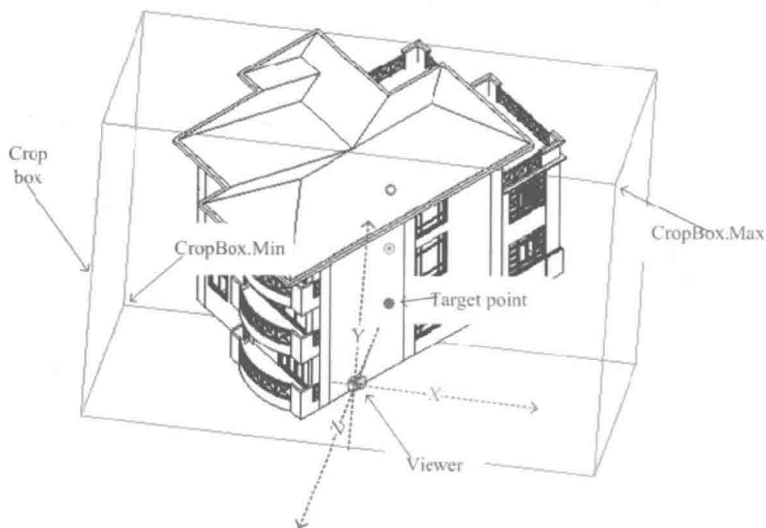


图 2-15 平行投影

模型投射到视图平面, 然后缩放到屏幕上, 见图 2-16。`View.Scale` 属性表示实际模型尺寸与视图尺寸的比例。相关表达式如下:



图 2-16 缩放视图平面窗口到屏幕视口

$\text{View.CropBox.Max.X (Y) / View.OutLine.Max.X (Y) = View.CropBox.Min.X (Y) / View.OutLine.Min.X (Y) = \text{View.Scale}$

#### 代码 2-48: View3D.CreateIsometric()方法

```
public static View3D View3D.CreateIsometric (Document document, ElementId viewFamilyTypeId;
```

ViewFamilyTypeId 需要是三维 ViewType。Revit 规定如下:

- 观察着位置。
- 如何用视图方向创建视图坐标系。
- 如何创建裁剪模型的裁剪框。

创建视图之后,可以调整裁剪框,以查看模型的不同部分。开发人员还可以更改默认方向。但 API 不支持修改视图坐标系。

代码 2-49 阐释了如何创建正等轴测三维视图。

#### 代码 2-49: 创建正等轴测三维视图

```
// Find a 3D view type
IEnumerable<ViewFamilyType> viewFamilyTypes = from elem in newFilteredElementCollector (document) .OfClass (typeof
(ViewFamilyType))
    let type = elem as ViewFamilyType
    where type.ViewFamily = ViewFamily.ThreeDimensional
    select type;

// Create a new View3D
View3D view3D = View3D.CreateIsometric (document, viewFamilyTypes.First().Id);
if (null != view3D)
{
    // By default, the 3D view uses a default orientation.
    // Change the orientation by creating and setting a ViewOrientation3D
    XYZ eye = new XYZ (10, 10, 10);
    XYZ up = new XYZ (0, 0, 1);
    XYZ forward = new XYZ (0, 1, 0);

    ViewOrientation3D viewOrientation3D = new ViewOrientation3D (eye, up, forward);
    view3D.SetOrientation (viewOrientation3D);
}
```





(3) 三维视图剖面框 (3D Views SectionBox)。每个视图都有一个裁剪框。裁剪框集中在模型的某一部分，投射并显示在视图中，见图 2-17。对于三维视图，还有另一个框，称为剖面框。

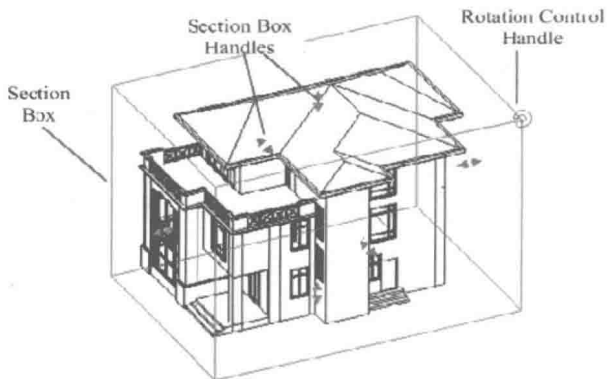


图 2-17 剖面框

- 剖面框确定在三维视图中显示模型的哪个部分。
- 剖面框用于剪辑三维模型的可见部分。
- 剖面框外的部分是不可见的，即使它在裁剪框内。
- 剖面框不同于裁剪框，它可以随模型旋转和移动。

对于大型模型，剖面框特别有用。例如，如果要渲染一个大型建筑，用剖面框。剖面框削减了需要计算的模型规模。要显示剖面框，请在三维视图图元属性对话框中范围部分，选择剖面框。也可以用 API 进行设置，见代码 2-50。

#### 代码 2-50：显示剖面框

```
private void ShowHideSectionBox (Autodesk.Revit.DB.View3D view3D)
{
    foreach (Parameter p in view3D.Parameters)
    {
        // Get Section Box parameter
        if (p.Definition.Name.Equals ("Section Box"))
        {
            // Show Section Box
            p.Set (1);
            // Hide Section Box
            // p.Set (0);
            break;
        }
    }
}
```

View3D.SectionBox 属性用于获取和更改框的范围。在某些情况下，设置 View3D.SectionBox 可能会产生意外结果。将该属性设置为某个值可以更改框的容量，并将其显示在视图中。也可以指定一个 null 值给剖面框来保存已修改的值并隐藏视图中的剖面框。为



避免显示剖面框，请更改该剖面框的值，然后设置该框为 null。代码 2-51 阐释了此过程。请注意，只有在视图属性对话框中选中剖面框的复选框时，它才有效。

**代码 2-51：隐藏剖面框**

```
private void ExpandSectionBox (View3D view)
{
    // The original section box
    BoundingBoxXYZ sectionBox = view.SectionBox;

    // Expand the section box
    XYZ deltaXYZ = sectionBox.Max - sectionBox.Min;
    sectionBox.Max += deltaXYZ / 2;
    sectionBox.Min -= deltaXYZ / 2;

    //After resetting the section box, it will be shown in the view.
    //It only works when the Section Box check box is
    //checked in View property dialog.
    view.SectionBox = sectionBox;

    //Setting the section box to null will make it hidden.
    view.SectionBox = null;           // line x
}
```

注意：设置 view.SectionBox 为 null 和使用 SectionBox 参数来隐藏剖面框具有同样效果。

从 SectionBox 参数返回的 BoundingBoxXYZ 的最大和最小点坐标不是全局坐标。若要将这些坐标转换为全局坐标，则需通过从 BoundingBoxXYZ.Transform 属性得到的变换来转换点坐标，见代码 2-52。

**代码 2-52：最大和最小坐标转换为全局坐标**

```
private void ConvertMaxMinToWCS (View3D view, out XYZ max, out XYZ min)
{
    BoundingBoxXYZ sectionbox = view.SectionBox;
    Transform transform = sectionbox.Transform;
    max = transform.OfPoint (sectionbox.Max);
    min = transform.OfPoint (sectionbox.Min);
}
```

(4) 视图锁定 (View Locking)。View3D 类具有与 Revit 用户界面中可用的锁定功能相对应的方法和属性，见图 2-18。



图 2-18 Revit 用户界面中可用的锁定功能选项



`View3D.SaveOrientationAndLock()` 方法可存储视图方向并锁定视图, 而 `View3D.RestoreOrientationAndLock()` 方法可恢复视图的方向, 然后将其锁定。如果当前视图已锁定, 可用 `View3D.Unlock` 解锁。`IsLocked` 属性将返回当前三维视图是否处于锁定状态。

### 3. 平面视图 (ViewPlan)

平面视图是基于标高的。有三种类型的平面视图, 即楼板平面视图、天花板平面视图和面积平面视图。

- 楼板平面视图通常是新项目打开时的默认视图。
- 大多数项目包括至少一个楼板平面视图和一个天花板平面视图。
- 在向项目添加新标高之后通常会创建平面视图。

用 API 添加新标高并不自动添加平面视图。使用静态方法 `ViewPlan.Create()` 来创建新的楼板和天花板平面视图。使用静态方法 `ViewPlan.CreateAreaPlan()` 来创建新的面积平面视图, 见代码 2-53。

#### 代码 2-53: 创建平面视图

```
public static ViewPlan ViewPlan.Create (Document document, ElementId viewFamilyTypeId, ElementId levelId);  
  
public static ViewPlan ViewPlan.CreateAreaPlan (Document document, ElementId areaSchemeId, ElementId levelId);
```

`ViewPlan.Create()` 方法的 `viewFamilyTypeId` 参数, 需要是 `FloorPlan`、`CeilingPlan`、`AreaPlan` 或 `StructuralPlan` 视图类型。`LevelId` 参数表示项目中与平面视图相关的标高图元的 ID。

代码 2-54 基于某个标高, 创建了一个楼板平面和一个天花板平面。

#### 代码 2-54: 创建楼板平面和天花板平面

```
// Find a floor plan view type  
IEnumerable<ViewFamilyType> viewFamilyTypes = from elem in new  
FilteredElementCollector (document) .OfClass (typeof (ViewFamilyType))  
let type = elem as ViewFamilyType  
where type.ViewFamily == ViewFamily.FloorPlan  
select type;  
  
// Create a Level and a Floor Plan based on it  
double elevation = 10.0;  
Level level1 = document.Create.NewLevel (elevation);  
ViewPlan floorView = ViewPlan.Create (document, viewFamilyTypes.First().Id, level1.Id);  
  
// Create another Level and a Ceiling Plan based on it  
// Find a ceiling plan view type  
viewFamilyTypes = from elem in new FilteredElementCollector (document) .OfClass (typeof (ViewFamilyType))  
let type = elem as ViewFamilyType  
where type.ViewFamily == ViewFamily.CeilingPlan  
select type;  
  
elevation += 10.0;  
Level level2 = document.Create.NewLevel (elevation);  
ViewPlan ceilingView = ViewPlan.Create (document, viewFamilyTypes.First().Id, level2.Id);
```



新建平面视图后,可以使用 **ViewDiscipline** 类型的 **Discipline** 参数来设置视图所涉规程。选项包括建筑、结构、机械、电气、水暖及协调。

对于结构平面视图,可使用 **ViewFamilyType.PlanViewDirection** 属性设置向上或向下的视图方向。虽然这是一个 **ViewFamilyType** 类的属性,但是除了 **StructuralPlan** 视图以外,如果从其他视图访问该属性则将引发异常。

通过 **ViewPlan.GetViewRange()** 方法可以检索平面视图的范围。**GetViewRange** 方法返回的 **PlanViewRange** 对象可以用来找出与平面相关的标高,以及各个平面对该标高的偏移量。这与 Revit 用户界面中 **View Range** 对话框中可得到的信息相同,见图 2-19。

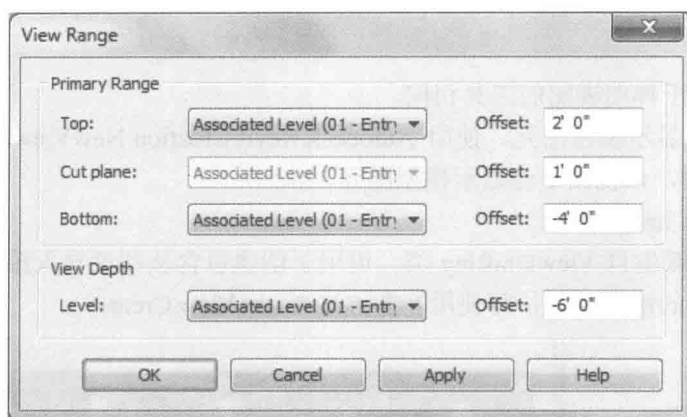


图 2-19 Revit 用户界面中 View Range 对话框中显示的信息

代码 2-55 演示了如何获取顶部剪辑平面及平面视图关联偏移量

#### 代码 2-55: 获取视图范围

```
private void ViewRange (Document doc, View view)
{
    if (view is ViewPlan)
    {
        ViewPlan viewPlan = view as ViewPlan;
        PlanViewRange viewRange = viewPlan.GetViewRange();

        ElementId topClipPlane = viewRange.GetLevelId (PlanViewPlane.TopClipPlane);
        double dOffset = viewRange.GetOffset (PlanViewPlane.TopClipPlane);

        if (topClipPlane.IntegerValue > 0)
        {
            Element levelAbove = doc.GetElement (topClipPlane);
            TaskDialog.Show (view.Name, "Top Clip Plane: " + levelAbove.Name + "\r\nTop Offset: " + dOffset + " ft");
        }
    }
}
```

#### 4. 绘图视图 (ViewDrafting)

绘图视图与模型是不相关的。它允许用户创建模型中未包括的详图。



- 绘图视图中，用户可以创建不同视图等级的详图（粗糙、精细或中等）。
- 可使用的二维详图工具包括：
  - Detail lines。
  - Reference planes。
  - Detail regions。
  - Dimensions。
  - Detail components。
  - Symbols。
  - Insulation。
  - Text。

这些工具和用于详图视图的工具相同。

- 绘图视图不显示模型图元。使用 Autodesk.Revit.Creation.NewViewDrafting()方法创建绘图视图，该视图不会显示模型图元。

#### 5. 图像视图 (ImageView)

ImageView 类派生自 ViewDrafting 类，可用于创建包含从硬盘导入图像的渲染视图，见图 2-20。创建新的渲染视图，可使用静态方法 ImageView.Create()。



图 2-20 Revit 用户界面中的图像视图

#### 6. 剖面视图 (ViewSection)

ViewSection 类可用于创建剖面视图、详图视图、详图索引、参照详图索引和参照剖面。它还可表示立面视图。

(1) 剖面视图和参照剖面 (Section Views and Reference Sections)。剖面视图剖开模型以显示其内部结构。剖面视图可用 ViewSection.CreateSection()方法创建，见代码 2-56。

##### 代码 2-56: ViewSection.CreateSection()方法

```
public ViewSection ViewSection.CreateSection (Document document, ElementId  
viewFamilyTypeId, BoundingBoxXYZ sectionBox);
```

viewFamilyTypeId 参数是个用于新剖面视图 viewFamilyType 的 ID。该类型须是一个剖面视图族。sectionBox 参数是剖面视图裁剪框。它提供了剖面视图所需的方向和范围。通常是将另一个视图裁剪框用作该参数。还可构建自定义 BoundingBoxXYZ 实例来表示方向和范围。

代码 2-57 演示了如何创建剖面视图。剖面视图的边界框创建在墙的中心位置。生成的剖面视图会置于项目浏览器中 Sections (Building Section) 节点位置。



### 代码 2-57: 创建剖面视图

```
// Find a section view type
IEnumerable<ViewFamilyType> viewFamilyTypes = from elem in new
FilteredElementCollector (document) .OfClass (typeof (ViewFamilyType))
    let type = elem as ViewFamilyType
    where type.ViewFamily == ViewFamily.Section
    select type;

// Create a BoundingBoxXYZ instance centered on wall
LocationCurve lc = wall.Location as LocationCurve;
Transform curveTransform = lc.Curve.ComputeDerivatives (0.5, true);
// using 0.5 and "true" (to specify that the parameter is normalized)
// places the transform's origin at the center of the location curve

XYZ origin = curveTransform.Origin; // mid-point of location curve
XYZ viewDirection = curveTransform.BasisX.Normalize(); // tangent vector along the location curve
XYZ normal = viewDirection.CrossProduct (XYZ.BasisZ) .Normalize(); // location curve normal @ mid-point

Transform transform = Transform.Identity;
transform.Origin = origin;
transform.BasisX = normal;
transform.BasisY = XYZ.BasisZ;

// can use this simplification because wall's "up" is vertical.
// For a non-vertical situation (such as section through a sloped floor the surface normal would be needed)
transform.BasisZ = normal.CrossProduct (XYZ.BasisZ);

BoundingBoxXYZ sectionBox = new BoundingBoxXYZ ();
sectionBox.Transform = transform;
sectionBox.Min = new XYZ (-10, 0, 0);
sectionBox.Max = new XYZ (10, 12, 5);
// Min & Max X values (-10 & 10) define the section line length on each side of the wall
// Max Y (12) is the height of the section box// Max Z (5) is the far clip offset

// Create a new view section.
ViewSection viewSection = ViewSection.CreateSection (document, viewFamilyTypes.First().Id, sectionBox);
```

参照剖面是一些参照现有视图的剖面。当创建一个新的参照剖面时, Revit 并不添加一个新视图, 见代码 2-58。

### 代码 2-58: ViewSection.CreateReferenceSection()方法

```
public ViewSection ViewSection.CreateReferenceSection (Document document,
    ElementId parentViewId,
    ElementId viewIdToReference,
    XYZ headPoint,
    XYZ tailPoint);
```

ParentViewId 参数是一个视图 ID, 该视图中显示新参照剖面标记。参照剖面可以在以



下视图中创建: 楼板平面、天花板平面、结构平面、剖面、立面、绘图、详图。viewIdToReference 可以是详图、绘图或剖面视图的 ID。新的参照剖面使用参照视图的 ViewFamilyType。两个空间点坐标将确定父视图中剖面标记符号的位置。

(2) 详图视图 (Detail Views)。详图视图是这样的模型视图, 表示另一视图中的详图索引或剖面。此类型的视图通常以比父视图更为精细的尺度表示模型的细节, 用于补充模型中指定部分的更多信息。静态方法 ViewSection.CreateDetail() 用于创建一个新的详细剖面视图, 见代码 2-59。

代码 2-59: ViewSection.CreateDetail() 方法

```
public ViewSection ViewSection.CreateDetail (Document document, ElementId  
viewFamilyTypeId, BoundingBoxXYZ sectionBox);
```

viewFamilyTypeId 参数是个用于新的剖面视图视图族类型的 ID。该类型须为详图视图族。正如标准剖面视图, sectionBox 参数表示剖面视图裁剪框, 规定了剖面视图所需的方向和范围。

添加新的详细剖面视图时, 它会出现项目浏览器中详图视图 (详图) 节点位置。

(3) 立面视图 (Elevation Views)。立面视图显示多根标高线, 表示模型的一个横剖面。立面视图是由 ViewSection 类来表示的, 见代码 2-60。然而, 与其他类型的剖面视图不同, 开发人员无法用 ViewSection 类的静态方法来创建立面视图。要创建一个立面视图, 首先要创建一个立面符号, 然后用此符号来生成立面视图。新创建的立面视图将出现在项目浏览器立面 (建筑立面) 节点位置。它会被指定一个唯一名称。

代码 2-60: 创建立面视图

```
ViewSection CreateElevationView (Document document, FamilyInstance beam)  
{  
    // Find an elevation view type  
    IEnumerable<ViewFamilyType> viewFamilyTypes = from elem in new FilteredElementCollector (document).OfClass (typeof  
(ViewFamilyType))  
    let type = elem as ViewFamilyType  
    where type.ViewFamily == ViewFamily.Elevation  
    select type;  
  
    LocationCurve lc = beam.Location as LocationCurve;  
    XYZ xyz = lc.Curve.GetEndPoint (0);  
    ElevationMarker marker = ElevationMarker.CreateElevationMarker (document, viewFamilyTypes.First().Id, xyz, 1);  
    ViewSection elevationView = marker.CreateElevation (document, document.ActiveView.Id, 1);  
  
    return elevationView;  
}
```

ElevationMarker.CreateElevation() 方法以 ViewPlan ID 作为参数。它是一个平面视图, 其中立面符号是可见的。新的立面视图将从 ViewPlan 派生其范围并继承其设置。最后一个参数是立面视图将被放置位置的立面符号索引。立面符号索引必须是有效的、未被使用过的。索引确定视图的方向。



(4) 详图索引和参照详图索引 (Callouts and Reference Callouts)。详图索引以更大的尺度显示另一个视图的一部分, 详图索引视图可以用静态方法 `ViewSection.CreateCallout()` 创建, 见代码 2-61。详图索引可由以下视图创建: 楼板平面、天花板平面、结构平面、剖面、立面、绘图、详图。生成的视图可以是剖面视图、平面视图或详图视图, 具体取决于所使用的 `ViewFamilyType`, 并会出现在项目浏览器中的对应节点上。

#### 代码 2-61: ViewSection.CreateCallout() 方法

```
public ViewSection ViewSection.CreateCallout (Document document,
                                             ElementId parentViewId,
                                             ElementId viewFamilyTypeId,
                                             XYZ point1,
                                             XYZ point2);
```

父视图 ID 参数可以是能够创建详图索引的任何类型视图的 ID。point 参数确定父视图中详图索引符号的范围。

参照详图索引是参照现有视图的详图索引, 如代码 2-62 所示创建了一个详图索引。当添加一个参照详图索引时, Revit 并不会在项目中创建视图, 而是创建指向指定的现有视图的指针。多个参照详图索引可以指向同一视图。

#### 代码 2-62: ViewSection.CreateReferenceCallout() 方法

```
public ViewSection ViewSection.CreateReferenceCallout (Document document,
                                                       ElementId parentViewId,
                                                       ElementId viewIdToReference,
                                                       XYZ point1,
                                                       XYZ point2);
```

创建参照详图索引类似于创建详图索引。但不同于详图索引有一个 `ViewFamilyType` ID 参数, `CreateReferenceCallout()` 方法是采用所参照视图的 ID。新建的参照详图索引会使用被参照视图的 `ViewFamilyType`。

只能参照裁剪过的视图, 除非参照的视图是绘图视图。不管父视图是何种类型, 总是可以参照绘图视图。从立面和绘图父视图可以参照立面视图, 从剖面和绘图父视图可以参照剖面视图。除楼板平面、天花板平面和结构平面父视图只能参照水平方向的详图视图之外, 可以从所有类型的父视图参照详图视图。从楼板平面、天花板平面和结构平面父视图可以参照楼板平面、天花板平面和结构平面视图。

代码 2-63 用详图视图族类型新建了一个详图索引, 然后使用该详图索引视图来创建一个参照详图索引。

#### 代码 2-63: 创建详图索引及参照详图索引

```
public void CreateCalloutView (Document document, View parentView)
{
    // Find a detail view type
    IEnumerable<ViewFamilyType> viewFamilyTypes = from elem in new
    FilteredElementCollector (document) .OfClass (typeof (ViewFamilyType))
    let type = elem as ViewFamilyType
    where type.ViewFamily == ViewFamily.Detail
```





```
select type;
```

```
ElementId viewFamilyTypeId = viewFamilyTypes.First().Id;
XYZ point1 = new XYZ (2, 2, 2);
XYZ point2 = new XYZ (30, 30, 30);
ElementId parentViewId = parentView.Id;    // a ViewPlan
View view = ViewSection.CreateCallout (document, parentViewId, viewFamilyTypeId, point1, point2);

ViewSection.CreateReferenceCallout (document, parentViewId, view.Id, point1, point2);
}
```

## 7. 图纸视图 (ViewSheet)

一张图纸包含视图和标题栏。在创建图纸视图时,使用 `ViewSheet.Create()` 方法。标题栏族符号 ID 是该方法的必需参数。`Autodesk.Revit.Document.TitleBlocks` 属性包含文件中的所有标题栏,在其中选择一个标题栏来创建图纸:

```
public static ViewSheet ViewSheet.Create ( Document document , ElementId
titleBlockTypeId);
```

新创建的图纸没有视图。`Viewport.Create()` 方法用于添加视图,见代码 2-64。`Viewport` 类用于将常规视图(即平面、立面、绘图和三维视图)添加到图纸视图。若要在视图中添加明细表,则需使用 `ScheduleInstance.Create()`。

### 代码 2-64: 添加视图到图纸

```
public static Viewport Viewport.Create (Document document, ElementId viewSheetId,
ElementId viewId, XYZ point);
```

- XYZ 位置参数标识添加的视图位于何处,其值为该视图的中心点坐标(单位为英寸)。
- 坐标 (0, 0) 对应于图纸的左下角。

每张图纸都有其完整图集中唯一的编号,即显示在项目浏览器中图纸名称之前的那个数字。这便于用户能使用视图标题中的图纸编号来交叉引用图集里的图纸。可以使用 `SheetNumber` 属性来检索或修改编号。编号必须是唯一的;否则,若编号设置为重复的值,将引发异常。

代码 2-65 阐释了如何创建并打印图纸视图。首先在文件中找出一个可用标题栏(可使用过滤器解决),并用它创建图纸视图。接着,可以添加三维视图,该视图左下角位置对应于图纸中心。最后,通过调用 `View.Print()` 方法打印图纸。

### 代码 2-65: 创建图纸视图

```
private void CreateSheetView (Autodesk.Revit.DB.Document document, View3D view3D)
{
    // Get an available title block from document
    IEnumerable<FamilySymbol> familyList = from elem in new FilteredElementCollector (document)
                                           .OfClass (typeof (FamilySymbol))
```



```

.OfCategory (BuiltInCategory.OST_TitleBlocks)
let type = elem as FamilySymbol
where type.Name.Contains ("E1")
select type;

// Create a sheet view
ViewSheet viewSheet = ViewSheet.Create (document, familyList.First().Id);
if (null == viewSheet)
{
    throw new Exception ("Failed to create new ViewSheet.");
}

// Add passed in view onto the center of the sheet
if (Viewport.CanAddViewToSheet (document, viewSheet.Id, view3D.Id))
{
    BoundingBoxUV sheetBox = viewSheet.Outline;
    double yPosition = (sheetBox.Max.V - sheetBox.Min.V) / 2 + sheetBox.Min.V;
    double xPosition = (sheetBox.Max.U - sheetBox.Min.U) / 2 + sheetBox.Min.U;

    XYZ origin = new XYZ (xPosition, yPosition, 0);
    Viewport viewport = Viewport.Create (document, viewSheet.Id, view3D.Id, origin);
}

// Print the sheet out
if (viewSheet.CanBePrinted)
{
    TaskDialog taskDialog = new TaskDialog ("Revit");
    taskDialog.MainContent = "Print the sheet?";
    TaskDialogCommonButtons buttons = TaskDialogCommonButtons.Yes | TaskDialogCommonButtons.No;
    taskDialog.CommonButtons = buttons;
    TaskDialogResult result = taskDialog.Show();

    if (result == TaskDialogResult.Yes)
    {
        viewSheet.Print();
    }
}
}

```

注意：不能添加某个图纸视图到另一图纸，并且不能添加视图到多个图纸，否则会出现参数异常。

打印机设置 (Printer Setup)：打印图纸前用户可能需要更改打印机设置。API 公开了与打印机设置相关的 **PrintManager** 类，以及一些相关的 **Autodesk.Revit.DB** 类，见表 2-17。

表 2-17 API 公开的与打印机设置相关的 **PrintManager** 类

类 名	功 能
Autodesk.Revit.DB.PrintManager	表示 Revit 用户界面里的打印对话框 (文件->打印) 中的打印信息



续表

类 名	功 能
Autodesk.Revit.DB.PrintParameters	包含文档打印设置的一个对象
Autodesk.Revit.DB.PrintSetup	表示 Revit 用户界面里的打印设置 (文件->打印设置...)
Autodesk.Revit.DB.PaperSize	一个对象, 表示 Autodesk Revit 项目打印设置中的纸张规格
Autodesk.Revit.DB.PaperSizeSet	一组可以包含任意个纸张规格的对象集
Autodesk.Revit.DB.PaperSource	一个对象, 表示 Autodesk Revit 项目打印设置中的纸张来源
Autodesk.Revit.DB.PaperSourceSet	一组可以包含任意种纸张来源的对象集
Autodesk.Revit.DB.ViewSheetSetting	表示 Revit 用户界面里的视图/页面设置 (文件->打印)
Autodesk.Revit.DB.PrintSetting	表示 Revit 用户界面里的打印设置 (文件->打印设置...)

有关使用这些对象的代码, 请参见 Revit 平台 SDK 中包含的 `ViewPrinter` 示例应用程序。

#### 8. 明细表视图 (ViewSchedule)

明细表以表格形式显示数据。典型的明细表会显示各种图元 (门、房间等), 每一行表示一个图元, 每一列代表一个参数。

`ViewSchedule` 类表示明细表和其他类似于明细表的视图, 包括单一类别和多类别明细表、关键字明细表、材料清单、视图列表、图纸列表、注释记号图例、修订明细表以及注释块。

`ViewSchedule.Export()` 方法将明细表数据导出到文本文件中。

(1) 将明细表置入图纸 (Placing Schedules on Sheet)。静态方法 `ScheduleSheetInstance.Create()` 会在图纸中创建一个明细表实例。它需要明细表所在图纸的图纸 ID、明细表视图 ID, 以及明细表所在图纸的 XYZ 定位。`ScheduleSheetInstance` 对象提供属性以访问生成此对象的“母”明细表的 ID、明细表在图纸内的旋转、明细表的放置位置 (以图纸坐标), 以及一个标志, 识别 `ScheduleSheetInstance` 是否为一个标题块族的修订明细表。

(2) 创建明细表 (Creating a Schedule)。`ViewSchedule` 类有几个根据明细表类型创建新明细表的方法。所有方法都有个 `Document` 参数, 它是将要添加明细表或类似明细表视图的目标文件。新创建的明细表视图会显示在项目浏览器的 `Schedules/Quantities` 节点下。

静态方法 `ViewSchedule.CreateSchedule()` 可以创建标准的单一类别或多类别明细表, 见代码 2-66。

#### 代码 2-66: ViewSchedule.CreateSchedule() 方法

```
public ViewSchedule ViewSchedule.CreateSchedule (Document document, ElementId categoryId);
```

`ElementId` 参数是类别 ID, 其图元将被列入明细表, 而对于多类别明细表, 它是非法图元 ID。

代码 2-67 介绍用 `CreateSchedule()` 方法创建一个面积明细表, 需要一个额外的参数: 该明细表的面积明细表 ID。



### 代码 2-67: 创建面积明细表

```
FilteredElementCollector collector1 = new FilteredElementCollector (doc);
collector1.OfCategory (BuiltInCategory.OST_AreaSchemes);
//Get first ElementId of AreaScheme.
ElementId areaSchemeId = collector1.FirstElementId();

//If you want to create an area schedule, you must use CreateSchedule method with three arguments.
//The input of second argument must be ElementId of BuiltInCategory.OST_Areas category and the input of third argument must
be ElementId of a areaScheme.
ViewSchedule areaSchedule = Autodesk.Revit.DB.ViewSchedule.CreateSchedule ( doc , newElementId ( BuiltInCategory.
OST_Areas), areaSchemeId);
```

关键字明细表显示抽象的“key”图元，用于填入普通模型图元的参数，可以用静态方法 `ViewSchedule.CreateKeySchedule()` 创建，其第二个参数是图元类别 ID，该明细表的关键字与之关联。

材料清单明细表显示构成模型图元的材料信息。与常规明细表每一行（分组前）代表单个图元不同，材料清单明细表中每一行代表单个<图元，材料>对。`ViewSchedule.CreateMaterialTakeoff()`方法与 `ViewSchedule.CreateSchedule()`方法具有相同的参数，可创建单一类别或多类别材料清单明细表。

视图列表、图纸列表和注释记号图例是与指定类别相关联的，因此它们的创建方法需要类别 ID 作为参数。视图列表是项目中的视图明细表，属于 `Views` 类别的明细表，用 `ViewSchedule.CreateViewList()`方法创建。图纸列表是项目中的图纸明细表，属于 `Sheets` 类别的明细表，用 `ViewSchedule.CreateSheetList()`方法创建。注释记号图例是 `Keynote Tags` 类别的明细表，用 `ViewSchedule.CreateKeynoteLegend()`方法创建。

修订明细表被添加到 `titleblock` 族，成为图纸中可见标题块的一部分。若传入的文档不是 `titleblock` 族，则 `ViewSchedule.CreateRevisionSchedule()`方法将引发异常。

注释块是 `Generic Annotations` 类别的明细表，它显示属于单个族的一些图元，而不是某个类别中的所有图元，见代码 2-68 和代码 2-69。

### 代码 2-68: `ViewSchedule.CreateNoteBlock()`方法

```
public ViewSchedule ViewSchedule.CreateNoteBlock (Document document, ElementId familyId);
```

`ElementId` 参数是族 ID，其图元将被列入明细表。

### 代码 2-69: 创建注释块明细表

```
//Get first ElementId of AnnotationSymbolType families.
ElementId annotationSymbolTypeId = ElementId.InvalidElementId;
if (!doc.AnnotationSymbolTypes.IsEmpty)
{
    foreach (AnnotationSymbolType type in doc.AnnotationSymbolTypes)
    {
        annotationSymbolTypeId = type.Family.Id;
        break;
    }
}
```



```
}  
  
//Create a noteblock view schedule.  
ViewSchedule noteBlockSchedule = ViewSchedule.CreateNoteBlock (doc, annotationSymbolTypeId);
```

(3) 使用明细表视图 (Working with ViewSchedule)。ScheduleDefinition 类包含各种设置, 定义明细表视图中的内容, 包括:

- 明细表类别和确定其类型的其他基本属性。
- 将成为明细表各列的一组字段。
- 分类和分组条件。
- 限制明细表中图元集是否可见的过滤器。

大多数明细表包含通过 ViewSchedule.Definition 属性检索到的单个 ScheduleDefinition。在 Revit MEP 中, 某些类别的明细表可以包含一个“嵌入式明细表”, 含有与主明细表图元关联的图元, 例如, 房间明细表显示每个房间内都有的图元, 或风管系统明细表显示与每个系统都相关的图元。嵌入式明细表有其自己的类别、字段、过滤器等。这些设置存储在第二个 ScheduleDefinition 对象中, 需要时, 可从 ScheduleDefinition.EmbeddedDefinition 属性获取嵌入式 ScheduleDefinition。

1) 添加字段 (Adding Fields)。创建视图明细表后即可添加字段。ScheduleDefinition.GetSchedulableFields() 方法将返回一个 SchedulableField 对象列表, 表示明细表中可包含的非计算字段。从 SchedulableField 对象或使用 ScheduleFieldType 枚举均可添加新字段。表 2-18 列出了从 ScheduleFieldType 枚举添加字段的可用选项。

表 2-18 从 ScheduleFieldType 枚举添加字段的可用选项

成员名称	说 明
Instance	明细表图元的实例参数。所有共享参数也同样使用此类型, 而不论它是实例还是类型参数
ElementType	明细表图元的类型参数
Count	明细表中, 显示在行内的图元数量
ViewBased	用于一些显示值可根据视图设置进行更改的参数的专门字段类型: <ul style="list-style-type: none"><li>• 房间和空间明细表中的 ROOM_AREA 和 ROOM_PERIMETER。</li><li>• 修订明细表中的 PROJECT_REVISION_REVISION_NUM。</li><li>• 按图纸进行编号的注释记号图例中的 KEYNOTE_NUMBER</li></ul>
Formula	来自明细表中其他字段值的计算公式
Percentage	百分比值, 表示另一个字段的每个图元占总数的百分比
Room	房间参数, 表示明细表图元属于该参数
FromRoom	房间参数, 依据“从”门旁或窗旁
ToRoom	房间参数, 依据“到”门旁或窗旁
ProjectInfo	明细表图元所属的项目中项目信息图元参数。它可以是个链接文件, 但仅当明细表中包含该链接文件中的图元时才可使用
Material	材料清单中的参数, 表示明细表图元的某种材料
MaterialQuantity	材料清单中的值, 表示明细表图元使用了多少特定材料。该参数 ID 可以是 MATERIAL_AREA、MATERIAL_VOLUME 或 MATERIAL_ASPAINT



续表

成员名称	说 明
RevitLinkInstance	链接文件中图元所属的 RevitLinkInstance 参数。目前 RVT_LINK_INSTANCE_NAME 是唯一受支持的参数, 只允许在包含链接文件图元的明细表中使用
RevitLinkType	链接文件中图元所属的 RevitLinkType 参数。目前 RVT_LINK_FILE_NAME_WITHOUT_EXT 是唯一受支持的参数, 只允许在包含链接文件图元的明细表中使用
StructuralMaterial	明细表图元的结构材料参数
Space	明细表图元所属的空间参数

使用 ScheduleDefinition.AddField()方法, 将字段添加到字段列表的末尾。若要将新字段放在字段列表中的特定位置, 则可使用 ScheduleDefinition.InsertField()方法之一。在使用 ScheduleDefinition.SetFieldOrder()后, 还可以对字段进行排序。

代码 2-70 说明了如何将字段添加到视图中, 若其尚未在明细表视图中。

**代码 2-70: 添加字段到明细表**

```

/// <summary>
/// Add fields to view schedule.
/// </summary>
/// <param name="schedules">List of view schedule.</param>
public void AddFieldToSchedule (List<ViewSchedule> schedules)
{
    IList<SchedulableField> schedulableFields = null;

    foreach (ViewSchedule vs in schedules)
    {
        //Get all schedulable fields from view schedule definition.
        schedulableFields = vs.Definition.GetSchedulableFields();

        foreach (SchedulableField sf in schedulableFields)
        {
            bool fieldAlreadyAdded = false;
            //Get all schedule field ids
            IList<ScheduleFieldId> ids = vs.Definition.GetFieldOrder();
            foreach (ScheduleFieldId id in ids)
            {
                //If the GetSchedulableField() method of gotten schedule field returns same schedulable field,
                // it means the field is already added to the view schedule.
                if (vs.Definition.GetField (id).GetSchedulableField() == sf)
                {
                    fieldAlreadyAdded = true;
                    break;
                }
            }

            //If schedulable field doesn't exist in view schedule, add it.
            if (fieldAlreadyAdded == false)
            {

```



```
vs.Definition.AddField (sf);  
    }  
    }  
}
```

**ScheduleField** 类表示字段 **ScheduleDefinition** 列表中的单个字段。每个（非隐藏）字段将成为明细表的一列。

最常见地，字段是表示出现在明细表中的图元的某个实例或类型参数。某些字段代表其他相关图元的参数，如某个明细表图元属于哪个房间。字段还可表示来自表中其他字段的计算数据、具体公式和百分比字段。

**ScheduleField** 类具有管理列标题的属性，包括文本及其方向。还可以定义在某一列内文本列宽和水平对齐方式。

**ScheduleField.IsHidden** 属性可用于隐藏字段。隐藏字段在明细表中是不会显示的，但它可用于过滤、排序、分组和条件格式化，还可被公式和百分比字段引用。

某些 **ScheduleFields** 可以进行总计，若 **HasTotals** 属性设置为 **true**，且启用了可显示总计的表尾行，则总计将显示在表尾行内。它可以是明细表末的总计行或明细表中成组字段之一的表尾行。在非逐项编号的明细表中，当多个图元出现在同一行时，总计也可显示在常规行内。

2) 字段样式和格式 (Style and Formatting of Fields)。**ScheduleField.GetStyle()** 和 **ScheduleField.SetStyle()** 使用 **TableCellStyle** 类来处理明细表中的字段样式。使用 **SetStyle()** 可以设置不同的字段属性，包括单元格的边框线条样式以及文本字体、颜色和大小。

**ScheduleField.SetFormatOptions()** 和 **ScheduleField.GetFormatOptions()** 使用 **FormatOptions** 类来处理字段的数据格式。**FormatOptions** 类包含一些设置，控制如何将带单位的数字格式化为字符串。它包含那些通常是由最终用户在格式对话框中选择并存储在文档中的设置。

在代码 2-71 中，所有明细表视图中的长度字段，转化成以英尺（小数部分以英寸）为单位来显示。

#### 代码 2-71：格式化一个字段

```
// format length units to display in feet and inches format  
public void FormatLengthFields (ViewSchedule schedule)  
{  
    int nFields = schedule.Definition.GetFieldCount();  
    for (int n = 0; n < nFields; n++)  
    {  
        ScheduleField field = schedule.Definition.GetField (n);  
        if (field.UnitType == UnitType.UT_Length)  
        {  
            FormatOptions formatOpts = new FormatOptions();  
            formatOpts.UseDefault = false;  
            formatOpts.DisplayUnits = DisplayUnitType.DUT_FEET_FRACTIONAL_INCHES;  
        }  
    }  
}
```



```

        field.SetFormatOptions (formatOpts);
    }
}

```

3) 明细表分组和排序 (Grouping and Sorting in Schedules)。ScheduleSortGroupField 类表示明细表可以以某个字段进行分组、排序。排序和分组是关联的操作。在任何情况下, 显示在明细表中的图元都是根据它们的字段值进行排序的, 明细表据此进行排序、分组, 自动将具有相同值的图元分在一组。通过启用附加的表头、表尾或空白行, 可以实现可视化的组间分隔。

如果 ScheduleDefinition.IsItemized 属性为 false, 则用于排序/分组的所有字段中, 具有相同值的图元将被合并到同一行; 否则, 明细表会在各自的行内显示每个图元。

对于使用 ScheduleField.IsHidden 属性把用于排序/分组的字段标记为隐藏的明细表, 也可以根据这些表中未显示的数据来进行排序、分组。

表头也可以分组。重载 ViewSchedule.GroupHeaders() 方法可用于指定哪些行、哪些列要包含在表头段分组中。其中一个重载方法需要所成组行和列的标题字符串。

在代码 2-72 中, 用标题将两个或更多的列成组。然后, 如果标题文本显示在列表头内, 再将它删除。

#### 代码 2-72: 标题分组

```

// Group columns of related data and remove redundant text from column headings
public void GroupRelatedData (ViewSchedule colSchedule, int startIndex, int endIndex, string groupText)
{
    colSchedule.GroupHeaders (0, startIndex, 0, endIndex, groupText);

    // If column heading has groupText in it, remove it
    // (i.e. if groupText is "Top" and field heading is "Top Level",
    // change the heading to just "Level"
    for (int index = startIndex; index <= endIndex; index++)
    {
        ScheduleField field = colSchedule.Definition.GetField (index);
        field.ColumnHeading = field.ColumnHeading.Replace (groupText, "");
    }
}

```

4) 过滤 (Filtering)。ScheduleFilter 可用于过滤将显示在明细表中的图元。过滤器是显示在明细表中的图元所必须满足的条件。显示在明细表中的图元必须满足所有过滤器。

对于使用 ScheduleField.IsHidden 属性把用于过滤的字段标记为隐藏的明细表, 也可以根据这些表中未显示的数据来进行过滤。

5) 使用明细表数据 (Working with Schedule Data)。ViewSchedule.GetTableData() 返回一个 TableData 对象, 该对象含有描述表格中行、列、单元格的样式和内容的大部分数据。更多信息详见 2.6.2 节表视图和表数据。





(4) 表视图和表数据 (TableView and TableData)。TableView 是 ViewSchedule 和 PanelScheduleView 的基类, 它是表示显示一张表格视图类。

1) 使用明细表数据 (Working with Data in a Schedule)。TableData 类中包含表格中的实际数据。虽然 TableData 对象不能直接从 TableView 类获得, 但两个子类都有 GetTableData() 方法。对于 ViewSchedule, 此方法返回一个 TableData 对象。而对于 PanelScheduleView, GetTableData() 返回一个从 TableData 基类派生的 PanelScheduleData 对象。TableData 类含有描述表格中行、列、单元格样式的大部分数据。PanelScheduleData 还提供专门与配电盘明细表相关的其他方法。

2) 使用行、列和单元格 (Working with Rows, Columns and Cells)。表格中的数据被分解成数段。要使用 TableData 的行、列和单元格, 必须先获取 TableSectionData 对象。TableData.GetSectionData() 可以用一个指向所需段数据的整数调用, 也可使用 SectionType (即 Header 或 Body) 来调用。

TableSectionData 类可用于插入或删除行或列, 设置单元格格式, 获取组成明细表本段内容的单元格详细数据, 如单元格类型 (即 Text 或 Graphic) 或者单元格的类别 ID。

在代码 2-73 中, 新行将添加到明细表的表头段, 并设置新建单元格的文本。请注意, 当从用户界面创建时, 表头段的第一行默认为标题。

#### 代码 2-73: 插入行

```
public void CreateSubTitle (ViewSchedule schedule)
{
    TableData colTableData = schedule.GetTableData ();

    TableSectionData tsd = colTableData.GetSectionData (SectionType.Header);
    tsd.InsertRow (tsd.FirstRowNumber + 1);
    tsd.SetCellText (tsd.FirstRowNumber + 1, tsd.FirstColumnNumber, "Schedule of column top and base levels with offsets");
}
```

注意: 只可在常规明细表表头段添加行和列。

另外还请注意, 在代码 2-73 中, 使用了属性 FirstRowNumber 和 FirstColumnNumber。在某些数据段中行或列的编号可能从 0 开始, 或者从 1 开始。应始终使用这些属性来替代编码 0 或 1。

明细表的行、列或单元格的样式可以自定义。这包括能够设置单元格的所有四个边框的线条样式, 以及单元格颜色和文本外观 (即颜色、字体、大小)。对于常规明细表, 只能在表格的表头段进行这些设置。

在代码 2-74 中, 明细表视图的副标题 (设定为标题段的第二行) 字体设置为粗体, 字号设置为 10 号。

#### 代码 2-74: 设置单元格格式

```
public void FormatSubTitle (ViewSchedule colSchedule)
{
    TableData colTableData = colSchedule.GetTableData ();
```



```

TableSectionData tsd = colTableData.GetSectionData (SectionType.Header);
// Subtitle is second row, first column
if (tsd.AllowOverrideCellStyle (tsd.FirstRowNumber + 1, tsd.FirstColumnNumber))
{
    TableCellStyle tcs = new TableCellStyle ();
    TableCellStyleOverrideOptions options = new TableCellStyleOverrideOptions ();
    options.FontSize = true;
    options.Bold = true;
    tcs.SetCellStyleOverrideOptions (options);
    tcs.IsFontBold = true;
    tcs.TextSize = 10;
    tsd.SetCellStyle (tsd.FirstRowNumber + 1, tsd.FirstColumnNumber, tcs);
}
}

```

### 2.6.3 视图过滤器 (View Filters)

使用 `ParameterFilterElement` 类, 可以将过滤器应用到视图。`ParameterFilterElement` 根据图元类别和一系列过滤规则来过滤图元。过滤器允许指定一个或多个类别。

定义过滤器 (一个或多个类别以及一项或多项过滤规则) 之后, 即可在几个方法中选择一个应用到视图中。`View.AddFilter()` 方法可将过滤器应用到视图, 但带有默认重载, 这意味着该视图的显示不会改变。`View.SetFilterOverrides()` 将设置与过滤器相关联的图形重载。`View.SetFilterVisibility()` 会设置通过过滤器的图元在视图中是否可见。如果尚未应用过滤器, 则 `AddFilter()` 和 `SetFilterVisibility()` 两者都可将其应用到视图, 从而不必单独调用 `AddFilter()`。

代码 2-75 创建了一个过滤器, 包括 `Comments` 属性设置为 "foo" 的所有墙壁。然后将过滤器应用到视图, 以便将满足这一条件的所有墙以红色示出其轮廓。

**代码 2-75: 过滤器应用到视图**

```

private void CreateViewFilter (Autodesk.Revit.DB.Document doc, View view)
{
    List<ElementId> categories = new List<ElementId>();
    categories.Add (new ElementId (BuiltInCategory.OST_Walls));
    ParameterFilterElement parameterFilterElement = ParameterFilterElement.Create (doc, "Comments = foo", categories);

    FilteredElementCollector parameterCollector = new FilteredElementCollector (doc);
    Parameter parameter = parameterCollector.OfClass (typeof (Wall)).FirstElement().get_Parameter ("Comments");

    List<FilterRule> filterRules = new List<FilterRule>();
    filterRules.Add (ParameterFilterRuleFactory.CreateEqualsRule (parameter.Id, "foo", true));
    parameterFilterElement.SetRules (filterRules);

    OverrideGraphicSettings filterSettings = new OverrideGraphicSettings ();
    // outline walls in red
    filterSettings.SetProjectionLineColor (new Color (255, 0, 0));
    view.SetFilterOverrides (parameterFilterElement.Id, filterSettings);
}

```



所有应用到视图的过滤器都可以使用 `View.GetFilters()` 方法检索, 其返回一个过滤器 ID 列表。对于某特定过滤器, 可分别使用 `View.GetFilterVisibility()` 和 `View.GetFilterOverrides()` 方法来检查过滤器的可见性和图形重载。 `View.RemoveFilter` 可用于从视图中删除过滤器。

#### 2.6.4 视图裁剪 (View Cropping)

某些视图的裁剪区可使用 Revit API 来修改。 `ViewCropRegionShapeManager.Valid` 属性指示是否允许操控视图裁剪区形状, 而 `ShapeSet` 属性指示形状是否已设置。代码 2-76 围绕房间的边界裁剪视图。

代码 2-76: 裁剪视图

```
public void CropAroundRoom (Room room, View view)
{
    if (view != null)
    {
        IList<IList<Autodesk.Revit.DB.BoundarySegment>> segments = room.GetBoundarySegments (new SpatialElement
        BoundaryOptions());

        if (null != segments) //the room may not be bound
        {
            foreach (IList<Autodesk.Revit.DB.BoundarySegment> segmentList in segments)
            {
                CurveLoop loop = new CurveLoop();
                foreach (Autodesk.Revit.DB.BoundarySegment boundarySegment in segmentList)
                {
                    loop.Append (boundarySegment.Curve);
                }

                ViewCropRegionShapeManager vcrShapeMgr = view.GetCropRegionShapeManager();
                vcrShapeMgr.SetCropRegionShape (loop);
                break; // if more than one set of boundary segments for room, crop around the first one
            }
        }
    }
}
```

#### 2.6.5 位移视图 (Displaced Views)

使用 `DisplacementElement` 类创建位移视图。 `DisplacementElement` 是个视图专有图元, 可用来使图元从它们的实际位置出现移位。位移视图可用于说明模型图元与整体模型之间的关系。 `DisplacementElement` 并不更改任何模型图元的实际位置, 仅仅是使它们显示在不同位置。

创建位移视图的详细例子, 请参阅 Revit SDK 中的 `DisplacementElementAnimation` 示例。

##### 1. 创建位移视图 (Creating a Displaced View)

使用静态方法 `DisplacementElement.Create()` 创建新位移图元。如果 `parentDisplacementElement` 参数不为 null, 则新位移图元可能是父位移图元的子图元。如果指定了父图元, 则子位移图元的变换将与其父图元的变换连接, 其关联图元的位移也与父位移图元相关。



`Create()`方法还需要文档、被移位图元的列表、所有者视图,以及应用到位移图元图形的转换。在任何视图中,某个图元只能通过单个位移图元移位。若指定某图元到多个位移图元则会引发异常。

位移图元的其他静态方法可以在调用 `Create()`之前使用,以利防止发生任何异常。`CanCategoryBeDisplaced()`测试图元是否属于可以位移的特定类别,而重载的静态方法 `CanElementsBeDisplaced()`指示某些特定图元可否被指定到一个新的位移图元。`IsAllowedAsDisplacedElement()`测试单个图元是否适合移位。

静态方法 `GetAdditionalElementsToDisplace()`将返回连同指定视图中的指定图元一起位移的任何其他图元。例如,当一堵墙移位时,任何嵌入或次生的图元也会被移位。

当创建子位移图元时,静态方法 `IsValidAsParentInView()`可以用来验证某个指定位移图元可否作为特定视图中的父位移图元。

位移图元的其他静态方法可用来寻找包括某个指定图元的位移图元,获取视图中所有位移图元的列表,或获取某指定视图所拥有的全部位移图元。

## 2. 使用位移图元 (Working with Displaced Elements)

创建新位移图元之后,即可使用方法获取任何子位移图元,获取所有受该位移图元影响的图元的 ID,或获取所有受该位移图元以及任何子位移图元影响的图元的 ID。只要有父图元的话, `ParentId` 属性还可返回父位移图元的图元 ID。

创建之后,可用 `SetDisplacedElementIds()`或 `RemoveDisplacedElement()`对受位移图元影响的图元集进行修改。此外,也可更改其相对位移。

`ResetDisplacedElements()`将设置位移图元平移到 (0, 0, 0)。位移图元依然存在,但其图元显示于它们的实际位置。

## 3. 创建位移路径 (Creating a Displaced Path)

位移路径是与位移图元相关的一个视图专有标记。`DisplacementPath` 类创建标记,描绘了图元从其实际位置向移置后位置的移动路径。通过参照位移图元要移置的图元边缘上的一个点,位移路径锚定位移图元上。它表示源自要移动图元上参照点的单一直线,或一系列的“折弯”线。

静态方法 `DisplacementPath.Create()`需要一个文档、位移图元的关联 ID、位移图元要移置图元某条边或曲线上的锚点,以及值域为 [0, 1] 的值,这是一个沿指定边缘的参数。一旦创建,位移图元的路径样式可以用 `PathStyle` 属性得到设置。锚点也可使用 `SetAnchorPoint()`作更改。

相关位移图元可能会有父位移图元,此父位移图元可能还会有其自己的父位移图元,由此产生一系列的被继承关系。其终点可能是点的原始(未移位的)位置,或对应于这些被继承位移图元任何中间位移位置上的相应点。`DisplacementPath.AncestorIdx` 属性用于指定路径的终点。

### 2.6.6 用户界面视图 (UIView)

Revit 中的 `View` 类是所有视图类型的基类并保持对视图中图元的跟踪,而 `UIView` 类包含与 Revit 用户界面视图窗口有关的数据。使用 `GetOpenUIViews()`方法,可以从



UIDocument 检索到所有打开视图的列表。UIView 类的方法可获取有关视图绘图区以及活动视图平移和缩放的信息。

UIView.GetWindowRectangle() 返回一个描述 UIView 窗口大小和位置的矩形。它不包括窗口边框和标题栏。

### 1. 缩放操作 ( Zoom Operations )

UIView 有一些有关活动视图缩放的方法。UIView.GetZoomCorners() 获取模型坐标系中视图矩形的四个角点坐标。而 UIView.ZoomAndCenterRectangle() 可将活动视图缩放、平移到模型输入区域的中心。

ZoomToFit() 和 ZoomSheetSize() 方法提供调整窗口缩放的快速方法, 而 Zoom() 方法可用于有指定比例因子的缩放。

### 2. 关闭视图 ( Closing a View )

UIView.Close() 用于关闭可视窗口。但它不能用来关闭最后一个活动窗口, 尝试关闭最后活动窗口将引发异常。

# 第 3 章 Revit 几何图元( Revit Geometric Elements )

## 3.1 墙、楼板、天花板、屋顶和洞口 ( Walls, Floors, Ceilings, Roofs and Openings )

本节讨论代表内置构筑物的图元和图元类型：

- HostObject: 主体对象，3.1.1 节和 3.1.2 节重点讨论主体对象以及相应的主体属性子类。
  - Foundation: 基础，在 API 中，不同的基础以不同的类表示，包括楼板、条形基础和族实例。在“楼板、天花板及基础”一节中会对它们进行比较。
  - CompoundStructure: 复核结构，该节描述 CompoundStructure 类并提供对材料的访问。某些主体图元类型具有热属性，在“热属性”一节中描述。
- 除主体图元之外，本节最后部分介绍了 Opening 类。

### 3.1.1 墙 ( Walls )

WallType.WallKind 枚举代表以下四种墙：

- Stacked (叠层墙)。
- Curtain (幕墙)。
- Basic (基础墙)。
- Unknown (未定义墙)。

Wall 和 WallType 类与 Basic 墙类型一起使用，提供有限的叠层墙和幕墙功能。有时需要检查某墙，以确定墙的类型。例如，不能用 API 从叠层墙获取其子墙。WallKind 是由系统族设置的只读枚举。

Wall.Flipped 属性和 Wall.flip()方法可访问并控制墙的方向。以下示例中，对调用 flip()方法之前和之后的墙作了对比。

- 在调用 flip()方法之前 Orientation 属性是 (0.0, 1.0, 0.0)。
- 在调用 flip()方法之后 Orientation 属性是 (0.0, -1.0, 0.0)。
- 墙定位线 (WALL\_KEY\_REF\_PARAM) 参数是 3 (详见表 3-1)。
- 以线作参考，墙移动了，但位置不变，见图 3-1 和图 3-2。

表 3-1 墙 的 定 位 线

定位线的值	说 明	定位线的值	说 明
0	墙中心线	1	核心中心线



续表

定位线的值	说 明	定位线的值	说 明
2	端面：外部	4	核心面：外部
3	端面：内部	5	核心面：内部



图 3-1 原墙



图 3-2 翻转后的墙

在 Wall 类中有五种静态重载方法来创建墙，详见表 3-2。

表 3-2 重载创建方法 Create()

名 称	说 明
Create (Document, Curve, WallType, Level, Double, Double, Boolean, Boolean)	指定高度、偏移量，在项目中新建一个指定类型的矩形轮廓墙
Create (Document, IList<Curve>, Boolean)	使用默认的墙类型，在项目创建非矩形轮廓的墙
Create (Document, Curve, ElementId, Boolean)	使用默认的墙类型，在项目于指定标高上根据图元 ID 新建矩形轮廓的墙
Create (Document, IList<Curve>, ElementId, ElementId, Boolean)	使用指定的墙类型，在项目创建非矩形轮廓的墙
Create (Document, IList<Curve>, ElementId, ElementId, Boolean, XYZ)	使用指定的墙类型和法向向量，在项目创建非矩形轮廓的墙

WallType 的墙功能 (WALL\_ATTR\_EXTERIOR) 参数 (表 3-3) 会影响所创建的墙实例 Room Bounding (房间边界) 和 Structural Usage (结构用法) 参数。WALL\_ATTR\_EXTERIOR 的值为整型。

表 3-3 墙 功 能

墙功能	内墙	外墙	基础墙	挡土墙	檐底板
Value	0	1	2	3	4

由 API 创建的墙遵循下列规则：

- 如果输入结构参数为 true 或 WallType 墙功能 (WALL\_ATTR\_EXTERIOR) 参数是 Foundation (基础)，那么墙结构用法参数是 Bearing (承重)；否则为 NonBearing (非承重)。
- 如果 WallType 墙功能 (WALL\_ATTR\_EXTERIOR) 参数是 Retaining (挡土墙)，则所创建的 Wall Room Bounding (WALL\_ATTR\_ROOM\_BOUNDING) 参数为 false。结构相关功能如 AnalyticalModel 属性的更多信息，请参阅 4.2 节 Revit Structure。



### 3.1.2 楼板、天花板和基础 ( Floors, Ceilings and Foundations )

与楼板、天花板和基础相关的 API 项目见表 3-4。

表 3-4 API 中的楼板、天花板和基础

对象	图元类型	图元类型的类型	图元创建	其他
Floor	Floor	FloorType	NewFloor( )/NewSlab( )	FloorType.IsFoundationSlab= false
Slab	Floor	FloorType	NewSlab( )	FloorType.IsFoundationSlab= false
Ceiling	Ceiling	CeilingType	No	Category = OST_Ceilings
Wall Foundation	ContFooting	ContFootingType	No	Category= OST_StructuralFoundation
Isolated Foundation	FamilyInstance	FamilySymbol	NewFamilyInstance( )	Category= OST_StructuralFoundation
Foundation Slab	Floor	FloorType	NewFloor( )	Category= OST_StructuralFoundation FloorType.IsFoundationSlab = true

注: Floor 和 Ceiling 由 CeilingAndFloor 类派生。

以下规则适用于楼板:

- 从基础设计栏创建的图元具有相同的类别属性, 即 OST\_StructuralFoundation, 但对应不同的类。
- FloorType IsFoundationSlab 属性设置 FloorType 是否为 OST\_StructuralFoundation 类别。

通过检索 FloorType 来创建楼板或基础板时, 请使用图 3-3 中的方法。

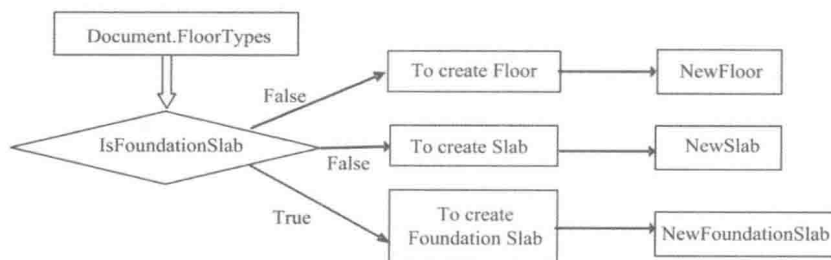


图 3-3 创建基础板和楼板/板

目前, API 不支持对 Floor 类的楼板坡度箭头的访问, 如图 3-4 但在 Revit Structure 中, 您可以代码 3-1 创建斜楼板。

#### 代码 3-1: NewSlab()

```
public Floor NewSlab (CurveArray profile, Level level, Line slopedArrow, double slope, bool isStructural);
```

使用 slopedArrow 参数可创建坡度箭头。

在 NewSlab() 中边坡参数的单位是坡高/坡水平宽度。

Floor.FloorType 属性可替代 Floor.GetTypeId() 方法。结构相关成员的更多信息, 如 GetSpanDirectionSymbolIds() 方法和 SpanDirectionAngle 属性, 请参阅 4.2 节 Revit Structure。

当在 Revit 中编辑独立基础时, 可以执行以下操作:

- 选择某个主体图元, 如楼板, 见图 3-5, 族实例对象的 Host 属性总是返回 null。



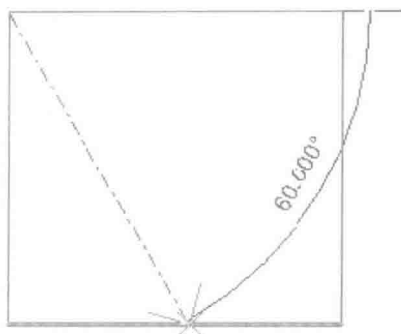


图 3-4 NewSlab 中的 slopedArrow 参数

- 当删除主体楼板时，基础并不随它一起删除。
- 基础主体可以从 Host (INSTANCE\_FREE\_HOST\_PARAM) 参数获得。
- 使用另一个有关联的 Offset 参数 (INSTANCE\_FREE\_HOST\_OFFSET\_PARAM) 可控制基础对主体图元的偏移。

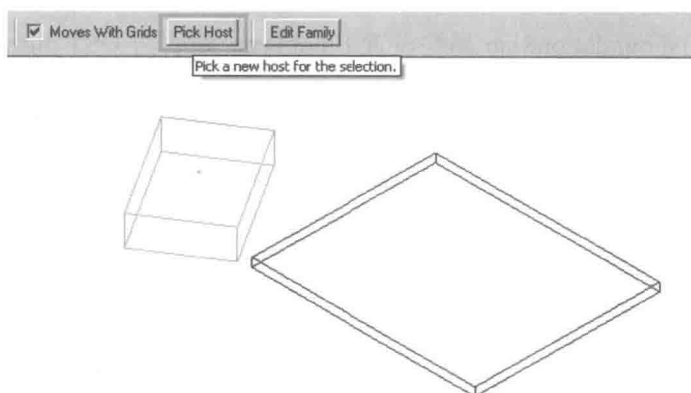


图 3-5 为基础板 (族实例) 选择主体图元

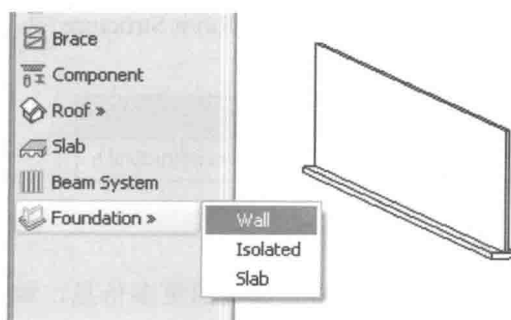


图 3-6 条形基础墙

在 API 中，条形基础是由 ContFooting 类表示的，见图 3-6。除使用 GetAnalytical Model()方法外（参考 4.2 节 Revit Structure 中的 AnalyticalModel），API 可对 ContFooting 和 ContFootingType 作受限访问。例如，在 Revit Architecture 中，附着墙不可用。Revit Structure 中，Wall 类与 ContFooting 类之间的关系是使用 AnalyticalModel 类中的 GetAnalyticalModelSupports()方法来表示的。有关详细信息，请参阅 4.2 节 Revit Structure 中

的 Analytical ModelSupport。



使用 `SlabShapeEditor` 类编辑基于板的图元形状，可以进行以下操作：

- 在基于板的所选图元上，处理一个或多个点、边缘。
- 在图元上添加点以更改图元的几何形状。
- 添加线性边缘，将现有板面分割成更小的子区域。
- 删除形状编辑器并重置图元几何形状为未修改的形状。

代码 3-2 示例了选择已作改动的楼板，还原其为修改前的原始形状。

**代码 3-2：还原板的形状**

```
private void ResetSlabShapes (Autodesk.Revit.DB.Document document)
{
    UIDocument uidoc = new UIDocument (document);
    Selection choices = uidoc.Selection;
    ElementSet collection = choices.Elements;
    foreach (Autodesk.Revit.DB.Element elem in collection)
    {
        Floor floor = elem as Floor;
        if (floor != null)
        {
            SlabShapeEditor slabShapeEditor = floor.SlabShapeEditor;
            slabShapeEditor.ResetSlabShape();
        }
    }
}
```

使用 `SlabShapeEditor` 和相关类的详细示例，请参阅 Revit SDK 中的 `SlabShapeEditing` 示例应用程序。

### 3.1.3 屋顶 (Roofs)

Revit 平台 API 中，所有 `Roofs` 都派生自 `RoofBase` 对象。有两个类：

- `FootPrintRoof`：迹线屋顶，表示屋顶由建筑物迹线生成。
- `ExtrusionRoof`：拉伸屋顶，表示屋顶由拉伸轮廓生成。

两者都有 `RoofType` 属性，该属性可读取或设置屋顶的类型。代码 3-3 演示了如何基于一些选定的墙创建迹线屋顶。

**代码 3-3：创建迹线屋顶**

```
// Before invoking this sample, select some walls to add a roof over.
// Make sure there is a level named "Roof" in the document.

// find the Roof level
FilteredElementCollector collector = new FilteredElementCollector (document);
collector.WherePasses (new ElementClassFilter (typeof (Level)));
var elements = from element in collector where element.Name == "Roof" select element;
Level level = elements.Cast<Level>().ElementAt<Level> (0);

RoofType roofType = null;
// select the first roofType
```



```
foreach (RoofType rt in document.RoofTypes)
{
    rooftype = rt;
    break;
}

// Get the handle of the application
Autodesk.Revit.ApplicationServices.Application application = document.Application;

// Define the footprint for the roof based on user selection
CurveArray footprint = application.Create.NewCurveArray();
UIDocument uidoc = new UIDocument (document);
if (uidoc.Selection.Elements.Size != 0)
{
    foreach (Autodesk.Revit.DB.Element element in uidoc.Selection.Elements)
    {
        Wall wall = element as Wall;
        if (wall != null)
        {
            LocationCurve wallCurve = wall.Location as LocationCurve;
            footprint.Append (wallCurve.Curve);
            continue;
        }

        ModelCurve modelCurve = element as ModelCurve;
        if (modelCurve != null)
        {
            footprint.Append (modelCurve.GeometryCurve);
        }
    }
}
else
{
    throw new Exception ("You should select a curve loop, or a wall loop, or loops combination \nof walls and curves to create a footprint roof.");
}

ModelCurveArray footPrintToModelCurveMapping = new ModelCurveArray();
FootPrintRoof footprintRoof = document.Create.NewFootPrintRoof ( footprint, level, rooftype, out footPrintToModelCurveMapping);
ModelCurveArrayIterator iterator = footPrintToModelCurveMapping.ForwardIterator();
iterator.Reset();
while (iterator.MoveNext())
{
    ModelCurve modelCurve = iterator.Current as ModelCurve;
    footprintRoof.set_DefinesSlope (modelCurve, true);
    footprintRoof.set_SlopeAngle (modelCurve, 0.5);
}
```



有关如何创建拉伸屋顶的示例，请参阅 Revit API SDK 中的 NewRoof 示例应用程序。

檐槽、封檐板是由代表屋顶 HostedSweep 类派生的图元。它们可以通过 API 创建、删除或修改。要创建这些图元，请使用重载 Document.Create. NewFascia() 或 Document. Create. NewGutter() 两者之一。如何新建檐槽和封檐板的例子，请参阅 Revit SDK 示例中含有的 NewHostedSweep 应用程序。代码 3-4 示例了如何修改檐槽图元的属性。

**代码 3-4: 修改檐槽**

```
public void ModifyGutter ( Autodesk.Revit.DB.Document document )
{
    UIDocument uidoc = new UIDocument ( document );
    ElementSet collection = uidoc.Selection.Elements;

    foreach ( Autodesk.Revit.DB.Element elem in collection )
    {
        if ( elem is Gutter )
        {
            Gutter gutter = elem as Gutter;
            // convert degrees to rads:
            gutter.Angle = 45.00 * Math.PI / 180;
            TaskDialog.Show ( "Revit", "Changed gutter angle" );
        }
    }
}
```

### 3.1.4 幕墙 ( Curtains )

幕墙、幕墙系统和幕墙屋顶均为 CurtainGrid (幕墙网格) 对象的主体图元。每个幕墙只能有一个幕墙网格，而幕墙系统和幕墙屋顶可能包含一个或多个幕墙网格。有关如何创建幕墙系统的示例，请参见 Revit SDK 中附带的 CurtainSystem 示例应用程序。如何创建幕墙并填充网格线的样例，请参阅 CurtainWallGrid 示例应用程序。

### 3.1.5 其他图元 ( Other Elements )

有些图元不是主体对象 (并且无特定的类)，但特殊情况下可以成为其他对象的主体。例如，坡道及其关联的图元类型，在 API 中没有特定的类，而是表示为 OST\_Ramps 类别中的图元和图元类型。

### 3.1.6 复合结构 ( CompoundStructure )

墙、楼板、天花板、屋顶都是主体对象的子类。主体对象 (及其相关类型的主体对象属性类) 提供对复合结构的只读访问。

CompoundStructure 类支持对含有不同材料组成的层集的读写访问：

- CompoundStructure.GetLayers()。
- CompoundStructure.SetLayers()。

这些层通常是平行的，并以固定层宽延伸整个主体对象。不过，对于墙体这种结构还可以“垂直复合”，这种情形下，层存在从墙顶部和底部特定的垂直距离的差异，通过 CompoundStructure.IsVerticallyCompound 方法来识别它们。对于垂直复合结构，结构垂直



剖面通过一个矩形来描述, 该矩形被分成侧边都是垂直或水平分段的多边形区域。映射将复合结构中每个区域与决定该区域属性的层索引关联。

使用复合结构, 可以找到各个层边界的几何位置。CompoundStructure. GetOffsetForLocationLine()方法可提供从中心定位线到任何定位线选项(核心中心线、任一端面或核心面)的偏移。

有了可用的定位线偏移量, 就可以使用 CompoundStructure.GetLayerWidth(), 从已知位置开始, 获取每个层边界的位置, 并获取各自的层宽。

使用复合结构需要注意以下事项:

- 图元的总宽度是每个复合结构层的宽度总和。虽然不能直接更改图元的总宽度, 但可以通过改变复合结构层的宽度来改变它。从 CompoundStructure. VariableLayerIndex, 可获取指定可变长度层(若已指定)索引。
- 必须设置复合结构返回到 HostObjAttributes 实例(使用 HostObjAttributes. SetCompoundStructure()方法), 以便存储所作的改变。
- HostObjAttributes 变化会影响当前文件中的每个实例。如果需要一个新的合并层, 则须使用 ElementType.Duplicate()方法新建一个 HostObjAttributes(), 并为其指定新的复合结构。
- 复合结构层 DeckProfileId、DeckEmbeddingType 属性仅适用于 Revit Structure 的板(Slab)。有关详细信息, 请参阅 Revit Structure。

在主体对象属性中, 每个复合结构层通常都会显示一些材料类型。如果 CompoundStructureLayer.MaterialId 返回-1, 则意味着材料是与类别相关的。有关详细信息, 请参阅 3.9 节材料。获取复合结构层材料, 见代码 3-5。

#### 代码 3-5: 获取复合结构层材料

```
public void GetWallLayerMaterial (Autodesk.Revit.DB.Document document, Wall wall)
{
    // get WallType of wall
    WallType aWallType = wall.WallType;
    // Only Basic Wall has compoundStructure
    if (WallKind.Basic == aWallType.Kind)
    {

        // Get CompoundStructure
        CompoundStructure comStruct = aWallType.GetCompoundStructure();
        Categories allCategories = document.Settings.Categories;

        // Get the category OST_Walls default Material;
        // use if that layer's default Material is <By Category>
        Category wallCategory = allCategories.get_Item (BuiltInCategory.OST_Walls);
        Autodesk.Revit.DB.Material wallMaterial = wallCategory.Material;

        foreach (CompoundStructureLayer structLayer in comStruct.GetLayers())
        {
            Autodesk.Revit.DB.Material layerMaterial =
```



```

        document.GetElement (structLayer.MaterialId) as Material;

        // If CompoundStructureLayer's Material is specified, use default
        // Material of its Category
        if (null == layerMaterial)
        {
            switch (structLayer.Function)
            {
                case MaterialFunctionAssignment.Finish1:
                    layerMaterial =
allCategories.get_Item (BuiltInCategory.OST_WallsFinish1) .Material;
                    break;
                case MaterialFunctionAssignment.Finish2:
                    layerMaterial =
allCategories.get_Item (BuiltInCategory.OST_WallsFinish2) .Material;
                    break;
                case MaterialFunctionAssignment.Membrane:
                    layerMaterial =
allCategories.get_Item (BuiltInCategory.OST_WallsMembrane) .Material;
                    break;
                case MaterialFunctionAssignment.Structure:
                    layerMaterial =
allCategories.get_Item (BuiltInCategory.OST_WallsStructure) .Material;
                    break;
                case MaterialFunctionAssignment.Substrate:
                    layerMaterial =
allCategories.get_Item (BuiltInCategory.OST_WallsSubstrate) .Material;
                    break;
                case MaterialFunctionAssignment.Insulation:
                    layerMaterial =
allCategories.get_Item (BuiltInCategory.OST_WallsInsulation) .Material;
                    break;
                default:
                    // It is impossible to reach here
                    break;
            }
            if (null == layerMaterial)
            {
                // CompoundStructureLayer's default Material is its SubCategory
                layerMaterial = wallMaterial;
            }
        }
        TaskDialog.Show ("Revit", "Layer Material: " + layerMaterial);
    }
}
}

```

有时仅需要“结构”层的材料，而不是在每个层中查找功能为 **MaterialFunction**



Assignment.Structure 的材料, 使用 CompoundStructure.StructuralMaterialIndex 属性来查找层索引, 其材料定义类型为分析用途的结构属性。

### 3.1.7 洞口 (Opening)

在 Revit 平台 API 中, 洞口对象派生自图元对象, 包含所有图元对象的属性和方法。要检索项目中的所有洞口, 可使用 Document.ElementIterator 找到 Elements.Opening 对象。

#### 1. 一般属性 (General Properties)

本节说明如何使用 Opening 属性。

- **IsRectBoundary:** 识别洞口是否为矩形边界。
  - 若为 true, 则意味着洞口有矩形边界, 可以从 Opening.BoundaryRect 属性获取一个 IList<XYZ> 集合。否则, 此属性返回 null。
  - 若为 false, 则会从 BoundaryCurves 属性获取一个 CurveArray 对象。
- **BoundaryCurves:** 若洞口边界为非矩形, 此属性检索几何形状信息; 否则返回 null。此属性返回一个含有表示洞口对象边界曲线的 CurveArray 对象。
- 关于曲线的更多细节, 请参阅 3.7 节。
- **BoundaryRect:** 如果洞口边界为矩形, 可以使用此属性获取几何信息; 否则它返回 null。
  - 此属性返回一个含有 XYZ 坐标的 IList<XYZ> 集合。
  - 该 IList<XYZ>集合通常含有矩形边界的最小 (左下角) 和最大 (右上角) 坐标。
- **Host:** 该属性检索洞口的主体图元。主体图元即被洞口对象切开的那个图元。

注意: 如果洞口对象的类别为 Shaft Openings (竖井洞口), 则洞口主体为 null。

代码 3-6 说明了如何检索现有洞口属性。

#### 代码 3-6: 检索现有洞口属性

```
private void Getinfo_Opening (Opening opening)
{
    string message = "Opening: ";

    //get the host element of this opening
    message += "\nThe id of the opening's host element is : " + opening.Host.Id.IntegerValue;

    //get the information whether the opening has a rect boundary
    //If the opening has a rect boundary, we can get the geometry information from BoundaryRect property.
    //Otherwise we should get the geometry information from BoundaryCurves property
    if (opening.IsRectBoundary)
    {
        message += "\nThe opening has a rectangular boundary.";
        //array contains two XYZ objects: the max and min coords of boundary
        IList<XYZ> boundaryRect = opening.BoundaryRect;

        //get the coordinate value of the min coordinate point
        XYZ point = opening.BoundaryRect [0] ;
        message += "\nMin coordinate point: (" + point.X + ", "
```



```

        + point.Y + ", " + point.Z + ") ";

        //get the coordinate value of the Max coordinate point
        point = opening.BoundaryRect [1];
        message += "\nMax coordinate point: (" + point.X + ", "
            + point.Y + ", " + point.Z + ") ";
    }
    else
    {
        message += "\nThe opening doesn't have a rectangular boundary.";
        // Get curve number
        int curves = opening.BoundaryCurves.Size;
        message += "\nNumber of curves is : " + curves;
        for (int i = 0; i < curves; i++)
        {
            Autodesk.Revit.DB.Curve curve = opening.BoundaryCurves.get_Item (i);
            // Get curve start point
            message += "\nCurve start point: " + XYZToString (curve.GetEndPoint (0));
            // Get curve end point
            message += ";   Curve end point: " + XYZToString (curve.GetEndPoint (1));
        }
    }
    TaskDialog.Show ("Revit", message);
}

// output the point's three coordinates
string XYZToString (XYZ point)
{
    return " (" + point.X + ", " + point.Y + ", " + point.Z + ") ";
}

```

## 2. 创建洞口 ( Create Opening )

在 Revit 平台 API 中, 使用 `Document.NewOpening()` 方法可在项目中创建洞口。在不同的主体图元上, 可以通过代码 3-7 所列的四种方法重载来创建洞口。

### 代码 3-7: NewOpening()

<pre> //Create a new Opening in a beam, brace and column. public Opening NewOpening (Element famInstElement, CurveArray profile, eRefFace iFace); </pre>
<pre> //Create a new Opening in a roof, floor and ceiling. public Opening NewOpening (Element hostElement, CurveArray profile, bool bPerpendicularFace); </pre>
<pre> //Create a new Opening Element. public Opening NewOpening (Level bottomLevel, Level topLevel, CurveArray profile); </pre>
<pre> //Create an opening in a straight wall or arc wall. public Opening NewOpening (Wall, XYZ pntStart, XYZ pntEnd); </pre>

- 在梁、斜撑或柱上创建洞口: 通常用族实例创建。iFace 参数指示洞口开在哪个





面上。

- 创建屋顶、楼板或天花板洞口：通常会在屋顶、楼板或天花板上创建洞口。
- **bPerpendicularFace** 参数指示洞口是否垂直于面或是竖向垂直的。若参数为 **true**，表示洞口垂直于主体图元面，见图 3-7 和图 3-8。

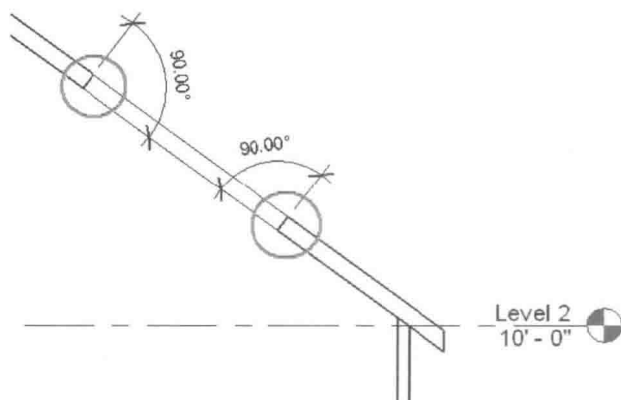


图 3-7 洞口切口垂直于主体图元面

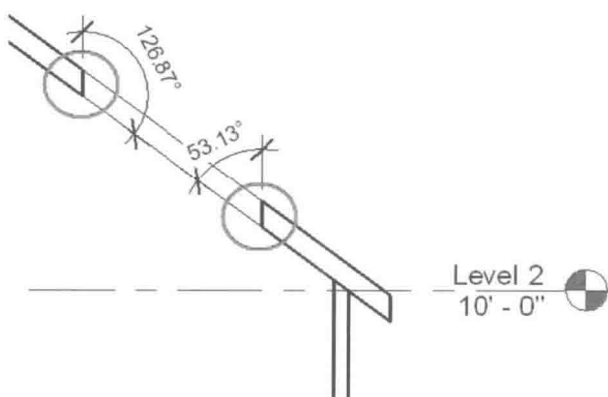


图 3-8 洞口竖向垂直切开主体图元

新建洞口图元：用来在项目中创建竖井洞口。但是，必须确保顶层高于底层，否则将引发异常。

在直墙或弧墙上创建洞口：用来在墙上创建矩形洞口。**pntStart** 和 **pntEnd** 坐标必须是能形成矩形的角点坐标。例如，矩形的左下角和右上角；否则将引发异常。

注意：使用洞口命令仅可创建矩形形状的墙洞。要在墙上创建其他形状的孔洞，须通过编辑墙的轮廓而不是使用洞口命令来实现。

### 3.1.8 热属性 (Thermal Properties)

某些组件类型，如墙壁、地板、天花板、屋顶和 **Building Pad** 都有计算出的以及可设置的热属性，由 **ThermalProperties** 类（图 3-9）表示。

**ThermalProperties** 类具有如图 3-9 所示的属性值。**Absorptance**（吸收率）和 **Roughness**



(粗糙度)可修改，而 HeatTransferCoefficient (热传导系数)、ThermalResistance (热阻)、ThermalMass (热体量) 是只读的，这些计算值的单位，见表 3-5。

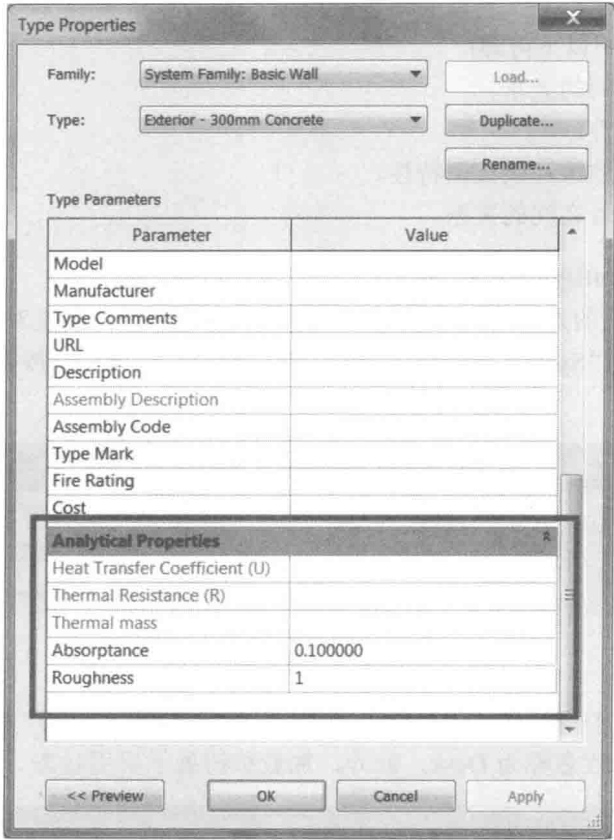


图 3-9 ThermalProperties 类

表 3-5 热属性计算值单位

属 性	单 位
HeatTransferCoefficient	瓦特每平方米凯尔文 [W/ (m <sup>2</sup> · K)]
ThermalResistance	平方米凯尔文每瓦特 [(m <sup>2</sup> · K) / W]
ThermalMass	千克平方英尺每秒平方凯尔文 [(kg · ft <sup>2</sup> ) / (s <sup>2</sup> · K)]

热属性可以使用以下类型的 ThermalProperties 属性来检索：

- WallType。
- FloorType。
- CeilingType。
- RoofType。
- BuildingPadType。



## 3.2 族实例 (Family Instances)

在本节中，将介绍以下内容：

- 族和族实例之间的关系。
- 族和族实例特性。
- 如何载入或创建族和族实例特性。
- 族实例和族符号之间的关系。

### 3.2.1 识别图元 (Identifying Elements)

在 Revit 中，判断图元是否为族实例，最简单的方法是使用属性对话框。

- 如果族名称以 “System Family” 开始（图 3-10），并且加载按钮被禁用，则它属于系统族。

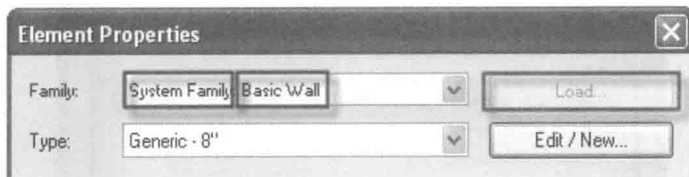


图 3-10 系统族

- 一般族实例属于构件族（图 3-11），名称不以 “System Family” 开头。例如，图 3-11 中桌子家具的族名称为 Desk，此外，加载按钮处于启用状态。

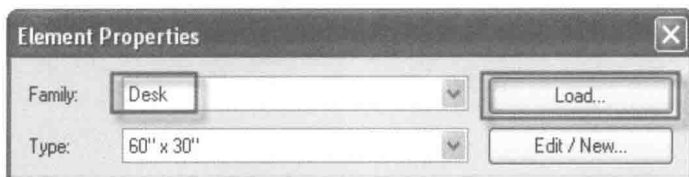


图 3-11 构件族

如图 3-12 所示有一些例外，如体量和内建成员，族和类型字段都是空白。

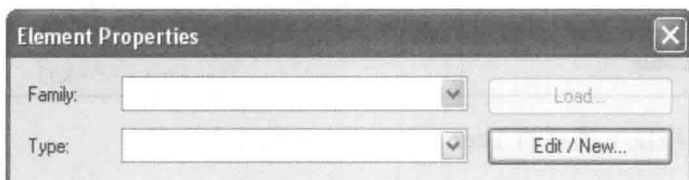


图 3-12 体量和内建成员示例

Revit 平台 API 中的族是由三个对象表示：

- 族。



- 族符号。
- 族实例。

族结构中的每个对象都有重要作用。

族对象表示整个族，如单齐平门。例如，单齐平门族对应 Single-Flush.rfa 文件。族对象包含多个 FamilySymbols，用于获取所有的族符号，以便符号之间的实例互换。

族符号对象表示对应于 Revit 用户界面中某个类型的一组特定的属性设置，如"34×80"。

族实例对象表示 Revit 项目中实际类型（族符号）的实例，如门的族实例。

- 每个族实例有一个族符号，门是一个"34×80"的实例。
- 每个族符号都属于一个族，此"34×80"符号属于一个单齐平门族。
- 每个族含有一个或多个族符号。单齐平门族含有"34×80"符号、"34×84"符号和"36×84"符号，等等。

注意：虽然大多数构件图元是通过 API 类 FamilySymbol 和 FamilyInstance 公开的，但有些已用特定的 API 类封装。例如，AnnotationSymbolType 封装 FamilySymbol，而 AnnotationSymbol 封装 FamilyInstance。

### 3.2.2 族 (Family)

族类代表整个 Revit 族，包括了族实例所使用的族符号。

#### 1. 载入族 (Loading Families)

Document 类包括 LoadFamily() 和 LoadFamilySymbol() 方法。

- LoadFamily() 可将整个族及其所有类型或符号加载到项目中。
- LoadFamilySymbol() 只将指定的族符号从族文件中加载到项目中。

注意：若要提高应用程序的性能和减少内存使用，则应尽可能地加载特定的族符号而不是整个族对象。

- 使用 Application 对象 Options 属性检索 Options.Application 对象。
- 使用 Options.Application 对象 GetLibraryPaths() 方法检索族文件路径。
- 在 LoadFamilySymbol() 中，输入的参数 Name 与 FamilySymbol 对象 Name 属性返回的字符串值相同。

#### 2. 类别 (Categories)

FamilyBase.FamilyCategory 属性指明族的类别，如柱、家具、结构框架或窗。

### 3.2.3 族实例 (FamilyInstances)

Revit 族实例对象类别的例子有梁、支撑、柱、家具、体量，等等。族实例对象具有更详细的属性，以便在项目中更改族实例的类型和外观。

#### 1. 位置相关属性 (Location-Related Properties)

位置相关属性显示族实例对象的物理和几何特征，如方向、旋转和位置。

(1) 方向 (Orientation)。对有些族实例对象，可以改变其面的朝向或把手的方向。例如，门可以面向房间或墙的外侧或内侧，把手可以放置在左侧或右侧。表 3-6 比较了门、窗和桌族实例。



表 3-6 族实例比较

布尔属性	门	窗 (固定窗: 36"w × 72"h)	桌
CanFlipFacing	true	true	false
CanFlipHand	true	false	false

如果 CanFlipFacing 或 CanFlipHand 为 true, 则 flipFacing()或 flipHand()方法可分别调用, 这些方法可分别更改面的朝向或把手方向; 否则, 这些方法不执行任何操作而返回一个 false。

需要注意的是, 改变方向时, 某些类型的窗可以更改把手方向和面朝向, 如 Trim 族的 3"×3"的 Casement (平开窗)。

门的朝向和把手方向有四种不同组合。参见图 3-13 的各种组合和表 3-7 对应的布尔值。

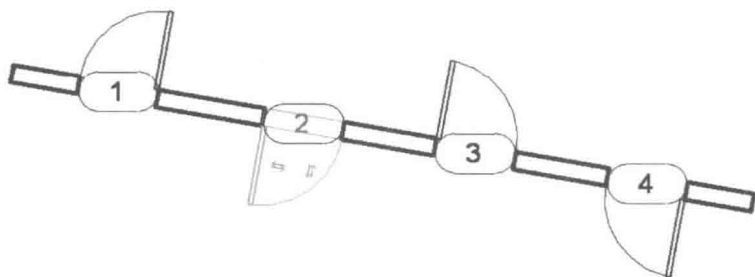


图 3-13 门的不同朝向和把手方向

表 3-7 同类型的不同实例

布尔属性	门 1	门 2	门 3	门 4
FacingFlipped	false	true	false	true
HandFlipped	false	true	true	false

(2) 方向-工作平面 (Orientation - Work Plane)。族实例的工作平面方向也是可以改变的。CanFlipWorkPlane 为 true 时, 可以设置 IsWorkPlaneFlipped 属性。如果在不允许工作平面翻转的族实例设置此属性, 则会引发异常。

(3) 旋转-镜像 (Rotation - Mirrored)。Mirrored 属性 (表 3-8) 指示族实例对象是否已被镜像。

表 3-8 门的镜像属性

布尔属性	门 1	门 2	门 3	门 4
Mirrored	false	false	true	true

在先前的门示例中, 门 1 和门 2 的镜像属性为 false, 而门 3 和门 4 都为 true。这是因为在 Revit 项目中创建一扇门时, 默认结果是门 1 或门 2。要创建像门 3 或门 4 这样的门, 必须分别翻转门 1 和门 2 的把手方向。翻转操作等同于镜像变换, 这就是门 3 和门 4 镜像属性为 true 的原因。

在 Revit 中使用 Mirror()方法的详细信息, 请参阅 2.5 节编辑图元。



(4) 旋转 - 可旋转和 Rotate() [Rotation - CanRotate and Rotate()]。族实例 CanRotate 布尔属性用于检查族实例是否可被旋转 180°。这取决于该实例所属的族。例如, 图 3-14 中的 CanRotate 属性: 窗 1 (3×3 平开窗: 36"×72") 和门 1 (双玻璃门 2: 72"×82") 为 true, 而窗 2 (固定窗: 36"w×72"h) 为 false。

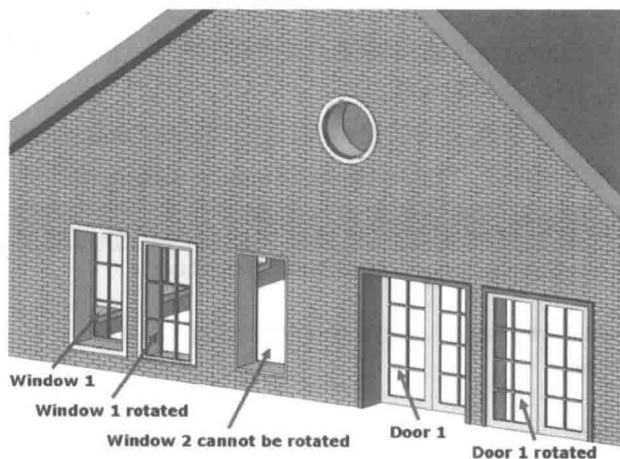


图 3-14 Rotate()后的变化

如果 CanRotate 为 true, 则可以调用族实例 Rotate() 方法, 将族实例翻转 180°。否则, 此方法不执行任何操作并返回 false。图 3-14 还显示了窗 1 和门 1 在执行 Rotate() 方法之后的状态。

回顾 2.5.3 节旋转图元所述, 使用 ElementTransformUtils.RotateElement() 和 ElementTransformUtils.RotateElements(), 那些族实例 (及其他图元) 可按用户指定的角度旋转。

(5) 位置 (Location)。Location 属性确定项目中实例的物理位置。实例可以有点位置或线位置。

适用于 Location 的特点如下:

- 点位置是个 LocationPoint 类对象 - 基础、门, 或有点位置的桌子。
- 线位置是个 LocationCurve 类对象 - 有线位置的梁。
- 它们都是 Location 类的子类。

有关位置的详细信息, 请参阅 2.5 节编辑图元。

## 2. 主体和主面 (Host and HostFace)

Host 和 HostFace 都是 FamilyInstance 属性。

(1) 主体 (Host)。族实例对象有个 Host 属性, 该属性返回其主体图元。

有些族实例对象没有主体图元, 如桌子和其他家具, 因为没有主体图元被创建, 所以 Host 属性不返回任何内容。然而, 其他对象, 如门窗, 必须有主体图元。在这种情况下, Host 属性返回窗或门所在的墙图元, 见图 3-15。

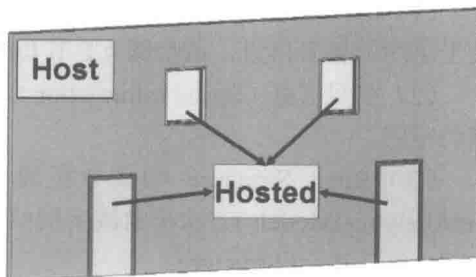


图 3-15 嵌于墙图元的门窗



(2) 主面 (HostFace)。HostFace 属性获取对族实例的主体表面的参照, 或若实例位于某个工作平面, 则获取对底部工作平面的几何面参照。若工作平面未参照其他几何图形, 或实例未嵌于某表面或工作平面, 则该属性返回值为空 (null)。

### 3. 子构件及父构件 (Subcomponent and Supercomponent)

FamilyInstance.GetSubComponentIds() 方法返回载入该族的族实例图元 ID。当一个 “Table-Dining Round w Chairs.rfa” 的实例置于某个项目中, 可由 GetSubComponentIds() 方法返回一组椅子图元的 ID, 见代码 3-8。

SuperComponent 属性返回族实例的父构件。在 “Table-Dining Round w Chairs.rfa” 中, 每个嵌入椅子的族实例父构件是 “Table-Dining Round w Chairs.rfa” 的实例。

代码 3-8: 由族实例获取子构件和父构件

```
public void GetSubAndSuperComponents (FamilyInstance familyInstance)
{
    ICollection<ElementId> subElemSet = familyInstance.GetSubComponentIds();
    if (subElemSet != null)
    {
        string subElems = "";
        foreach (Autodesk.Revit.DB.ElementId ee in subElemSet)
        {
            FamilyInstance f = familyInstance.Document.GetElement (ee) as FamilyInstance;
            subElems = subElems + f.Name + "\n";
        }
        TaskDialog.Show ("Revit", "Subcomponent count = " + subElemSet.Count + "\n" + subElems);
    }
    FamilyInstance super = familyInstance.SuperComponent as FamilyInstance;
    if (super != null)
    {
        TaskDialog.Show ("Revit", "SUPER component: " + super.Name);
    }
}
```

### 4. 其他属性 (Other Properties)

本节中的属性是特指 Revit Architecture 和 Revit Structure 中的属性。在它们各自的章节中会作具体介绍。

(1) 房间信息 (Room Information)。族实例属性包括 Room、FromRoom 和 ToRoom。关于房间的更多信息, 请参阅 4.1 节 Revit Architecture。

(2) 空间信息 (Space Information)。族实例有一个 Space 属性, 用于标识保存 MEP 实例的空间。

(3) Revit Structure 相关分析模型 (Revit Structure Related Analytical Model)。GetAnalyticalModel() 方法检索族实例结构分析模型。有关 AnalyticalModel 的更多信息, 请参阅 4.2 节 Revit Structure。

### 5. 创建族实例对象 (Creating FamilyInstance Objects)

通常, 创建族实例对象使用 Autodesk.Revit.Creation.Document 类的 NewFamilyInstance()



方法, 该方法具有 12 个重载方法。选择使用哪个重载不仅取决于实例的类别, 而且与其他位置特征也有关, 如该实例对象是否应嵌入主体、置于相关参照标高, 或直接置于特定的面。详情见表 3-9。

表 3-9 用 NewFamilyInstance() 创建实例的选项

类别	NewFamilyInstance() 参数	说 明
Air Terminals Boundary Conditions Casework Communication Devices Data Devices Electrical Equipment Electrical Fixtures Entourage Fire Alarm Devices Furniture Furniture Systems Generic Models Lighting Devices Lighting Fixtures Mass Mechanical Equipment Nurse Call Devices Parking Planting Plumbing Fixtures Security Devices Site Specialty Equipment Sprinklers Structural Connections Structural Foundations Structural Stiffeners Telephone Devices	XYZ, FamilySymbol, StructuralType	在无参照标高或主体图元的任意位置创建实例
	XYZ, FamilySymbol, Element, StructuralType	如果实例对象嵌于墙、楼板或天花板上
	XYZ, FamilySymbol, XYZ, Element, StructuralType	如果对象嵌于墙、楼板或天花板上, 并需要非默认的定位方向
	XYZ, FamilySymbol, Element, Level, StructuralType	如果对象嵌于墙、楼板或天花板上, 并与某个参照标高关联
	XYZ, FamilySymbol, Level, StructuralType	如果对象与某个参照标高关联
	Face, XYZ, XYZ, FamilySymbol	如果是基于面的对象, 并需要非默认的定位方向
	Reference, XYZ, XYZ, FamilySymbol	如果是基于面的对象, 并需要非默认的定位方向, 接受“面参照”而非某个面
	Face, Line, FamilySymbol	如果是基于面且是线性的对象
	Reference, Line, FamilySymbol	如果是基于面的线性对象, 但接受“面参照”而非某个面
Columns Structural Columns	XYZ, FamilySymbol, Level, StructuralType	创建基面位于参照标高上的柱。该柱将延伸到模型中邻近的可用标高, 或以默认柱高延伸 (若参照标高上方没有合适的标高)
Doors Windows	XYZ, FamilySymbol, Element, StructuralType	门窗必须嵌于墙上。若它们可被置于默认方向, 则使用此方法
	XYZ, FamilySymbol, XYZ, Element, StructuralType	如果所创实例需要一个非默认的定位方向
	XYZ, FamilySymbol, Element, Level, StructuralType	如果实例需要与一个参照标高关联
Structural Framing (Beams, Braces)	Curve, FamilySymbol, Level, StructuralType	根据给定曲线创建基于标高的支撑或梁。这是创建支撑或梁的推荐方法
	XYZ, FamilySymbol, StructuralType	在任意位置创建实例 <sup>①</sup>
	XYZ, FamilySymbol, Element, Level, StructuralType	如果实例嵌于某个图元 (楼板等) 并与参照标高关联 <sup>①</sup>
	XYZ, FamilySymbol, Level, StructuralType	若实例与参照标高关联 <sup>①</sup>
	XYZ, FamilySymbol, Element, StructuralType	如果实例嵌于某个图元 (楼板等) <sup>①</sup>
Detail Component	Line, FamilySymbol, View	仅适用于二维族基于直线的详图符号
Generic Annotations	XYZ, FamilySymbol, View	仅适用于二维族符号

① 结构实例创建后将为零长度。使用 LocationCurve.Curve 属性, 通过设置曲线 (FamilyInstance.Location 作为 LocationCurve) 来延伸。





一些 **FamilyInstance** 对象需要创建多个位置, 在这种情况下, 使用该对象所提供的更详细的创建方法更为适合。如果实例未创建, 则引发异常。调用该方法之前, 必须先将类型/符号加载到项目中。

使用 **Document.NewFamilyInstances()**, 通过一次创建多个族实例 (代码 3-9), 可以精简代码并提高性能。此方法有个单独的参数, 即描述所创建族实例的 **FamilyInstanceCreationData** 对象列表。

#### 代码 3-9: 成批创建族实例

```
ICollection<ElementId> BatchCreateColumns (Autodesk.Revit.DB.Document document, Level level)
{
    List<FamilyInstanceCreationData> fiCreationDatas = new List<FamilyInstanceCreationData>();

    ICollection<ElementId> elementSet = null;

    //Try to get a FamilySymbol
    FamilySymbol familySymbol = null;
    FilteredElementCollector collector = new FilteredElementCollector (document);
    ICollection<Element> collection = collector.OfClass (typeof (FamilySymbol) ).ToElements();
    foreach (Element e in collection)
    {
        familySymbol = e as FamilySymbol;
        if (null != familySymbol.Category)
        {
            if ("Structural Columns" == familySymbol.Category.Name)
            {
                break;
            }
        }
    }

    if (null != familySymbol)
    {
        //Create 10 FamilyInstanceCreationData items for batch creation
        for (int i = 1; i < 11; i++)
        {
            XYZ location = new XYZ (i * 10, 100, 0);
            FamilyInstanceCreationData fiCreationData = new FamilyInstanceCreationData (location, familySymbol, level,
                StructuralType.Column);
            if (null != fiCreationData)
            {
                fiCreationDatas.Add (fiCreationData);
            }
        }

        if (fiCreationDatas.Count > 0)
        {
            // Create Columns
            elementSet = document.Create.NewFamilyInstances2 (fiCreationDatas);
        }
    }
}
```



```

    }
    else
    {
        throw new Exception ("Batch creation failed.");
    }
}
else
{
    throw new Exception ("No column types found.");
}

return elementSet;
}

```

对于某些族类型的实例，通过表 3-10 的其他方法创建可能比 Autodesk.Revit. Creation.Document.NewFamilyInstance() 更好。

表 3-10 创建实例其他方法选项

类别	创建方法	说 明
Air Terminal Tags Area Load Tags Area Tags Casework Tags Ceiling Tags Communication Device Tags Curtain Panel Tags Data Device Tags Detail Item Tags Door Tags Duct Accessory Tags Duct Fitting Tags Duct Tags Electrical Equipment Tags Electrical Fixture Tags Fire Alarm Device Tags Flex Duct Tags Flex Pipe Tags Floor Tags Furniture System Tags Furniture Tags Generic Model Tags Internal Area Load Tags Internal Line Load Tags Internal Point Load Tags Keynote Tags Lighting Device Tags Lighting Fixture Tags Line Load Tags Mass Floor Tags Mass Tags Mechanical Equipment Tags Nurse Call Device Tags Parking Tags Pipe Accessory Tags Pipe Fitting Tags Pipe Tags Planting Tags Plumbing Fixture Tags Point Load Tags Property Line Segment Tags	NewTag (View, Element, Boolean, TagMode, TagOrientation, XYZ)	TagMode 应是 TM_ADDBY_CATEGORY, 在尝试创建标记时, 应该载入相关标记族, 否则将引发异常



续表

类别	创建方法	说 明
Property Tags Railing Tags Revision Cloud Tags Roof Tags Room Tags Security Device Tags Site Tags Space Tags Specialty Equipment Tags Spinkler Tags Stair Tags Structural Area Reinforcement Tags Structural Beam System Tags Structural Column Tags Structural Connection Tags Structural Foundation Tags Structural Framing Tags Structural Path Reinforcement Tags Structural Rebar Tags Structural Stiffener Tags Structural Truss Tags Telephone Device Tags Wall Tags Window Tags Wire Tags Zone Tags	NewTag (View, Element, Boolean, TagMode, TagOrientation, XYZ)	TagMode 应是 TM_ADDBY_CATEGORY, 在尝试创建标记时, 应该载入相关标记族, 否则将引发异常
Material Tags	NewTag (View, Element, Boolean, TagMode, TagOrientation, XYZ)	TagMode 应是 TM_ADDBY_MATERIAL, 并应载入相关材料标记族, 否则将引发异常
Multi-Category Tags	NewTag (View, Element, Boolean, TagMode, TagOrientation, XYZ)	TagMode 应是 TM_ADDBY_MULTICATEGORY, 并应载入一个多类别标记族, 否则将引发异常
Title Blocks	NewViewSheet (FamilySymbol)	标题块将被添加到新图纸

使用 Document.LoadFamily()或 Document. LoadFamilySymbol()方法加载族和族符号。有些族, 如梁, 有多个端点, 以同于单点实例的方式加插。线性族实例插入后, 可以通过 Element.Location 属性更改其端点。

3.2.4 代码示例 ( Code Samples )

NewFamilyInstance()方法需要设定 StructuralType 参数 (表 3-11), 以指定创建族实例的类型。以下是几个演示创建实例的例子。

表 3-11 在 NewFamilyInstance()方法中的 StructuralType 参数值

族实例类型	StructuralType 值	族实例类型	StructuralType 值
门、桌子等	NonStructural	柱	Column
梁	Beam	基础	Footing
支撑	Brace		

1. 创建桌子 ( Create Tables )

代码 3-10 演示了如何加载 Tables 族到 Revit 项目中, 并从该族的所有符号创建实例。

**代码 3-10: 创建桌子**

```
String fileName = @"C : \Documents and Settings\All Users\Application Data\Autodesk\RST 2011\Imperial
Library\Furniture\Table-Dining Round w Chairs.rfa";

// try to load family
Family family = null;
if (!document.LoadFamily (fileName, out family))
{
    throw new Exception ("Unable to load " + fileName);
}
// Loop through table symbols and add a new table for each
FamilySymbolSetIterator symbolItr = family.Symbols.ForwardIterator();
double x = 0.0, y = 0.0;
while (symbolItr.MoveNext())
{
    FamilySymbol symbol = symbolItr.Current as FamilySymbol;
    XYZ location = new XYZ (x, y, 10.0);
    // Do not use the overloaded NewFamilyInstance() method that contains
    // the Level argument, otherwise Revit cannot show the instances
    // correctly in 3D View, for the table is not level-based component.
    FamilyInstance instance = document.Create.NewFamilyInstance (location, symbol, StructuralType.NonStructural);

    x += 10.0;
}
```

若指定的族已被加载, 则 LoadFamily()方法返回 false。因此, 此代码被调用之前, 不能预先加载 “Table-Dining Round w Chairs.rfa”。本例代码默认情况下, 桌子创建于标高 1 上。

载入桌族并放置每个族符号的一个实例的结果见图 3-16。

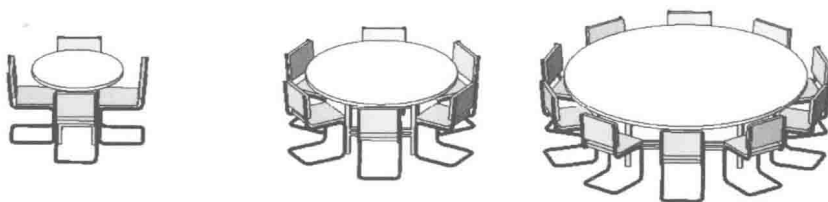


图 3-16 在 Revit 项目中载入族并创建桌子

## 2. 创建梁 (Create a Beam)

在代码 3-11 中, 载入族符号而不是族, 因为加载单个族符号要比加载包含多个族符号的族更快。

**代码 3-11: 创建梁**

```
// get the active view's level for beam creation
Level level = document.ActiveView.Level;

// load a family symbol from file
FamilySymbol gotSymbol = null;
```



```
String fileName = @"C : \Documents and Settings\All Users\Application Data\Autodesk\RST 2011\Imperial
Library\Structural\Framing\Steel\W-Wide Flange.rfa";
String name = "W10X54";

FamilyInstance instance = null;

if (document.LoadFamilySymbol (fileName, name, out gotSymbol))
{
    // look for a model line in the list of selected elements
    UIDocument uidoc = new UIDocument (document);
    ElementSet sel = uidoc.Selection.Elements;
    ModelLine modelLine = null;
    foreach (Autodesk.Revit.DB.Element elem in sel)
    {
        if (elem is ModelLine)
        {
            modelLine = elem as ModelLine;
            break;
        }
    }
    if (null != modelLine)
    {
        // create a new beam
        instance = document.Create.NewFamilyInstance (modelLine.GeometryCurve, gotSymbol, level, StructuralType.Beam);
    }
    else
    {
        throw new Exception ("Please select a model line before invoking this command");
    }
}
else
{
    throw new Exception ("Couldn't load " + fileName);
}
```

### 3. 创建门 (Create Doors)

在运行代码 3-12 之前, 先创建长度约 180' 的长墙并选择它。主体对象必须支持插入实例, 否则 `NewFamilyInstance()` 方法会失败。若未为实例提供主体图元, 则必须创建一个, 否则实例不能插入到指定的主体图元, `NewFamilyInstance()` 方法不执行任何操作。

#### 代码 3-12: 创建门

```
void CreateDoorsInWall (Autodesk.Revit.DB.Document document, Wall wall)
{
    String fileName = @"C: \Documents and Settings\All Users\Application Data\Autodesk\RST 2011\Imperial Library\Doors\Single-
Decorative 2.rfa";

    Family family = null;
    if (!document.LoadFamily (fileName, out family))
```



```

{
    throw new Exception ("Unable to load " + fileName);
}

// get the active view's level for beam creation
Level level = document.ActiveView.Level;

FamilySymbolSetIterator symbolItr = family.Symbols.ForwardIterator();
double x = 0, y = 0, z = 0;
while (symbolItr.MoveNext())
{
    FamilySymbol symbol = symbolItr.Current as FamilySymbol;
    XYZ location = new XYZ (x, y, z);
    FamilyInstance instance = document.Create.NewFamilyInstance (location, symbol, wall, level, StructuralType.NonStructural);
    x += 10;
    y += 10;
    z += 1.5;
}
}

```

代码 3-12 在 Revit 中的结果见图 3-17。请注意, 如果指定位置不在指定层上, 则 NewFamilyInstance() 方法使用该位置的标高而不是层标高。

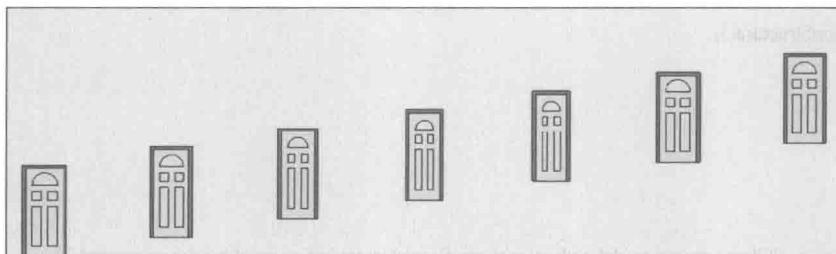


图 3-17 在墙上插入门

#### 4. 使用参照方向创建族实例 (Create FamilyInstances Using Reference Directions)

使用参照方向在特定的方向插入某个项, 见代码 3-13。

##### 代码 3-13: 使用参照方向创建族实例

```

String fileName = @"C : \Documents and Settings\All Users\Application Data\Autodesk\RST 2011\Imperial
Library\Furniture\Bed-Box.rfa";

Autodesk.Revit.DB.Family family = null;
if (!document.LoadFamily (fileName, out family))
{
    throw new Exception ("Couldn't load " + fileName);
}

FilteredElementCollector collector = new FilteredElementCollector (document);

```



```
Floor floor = collector.OfClass (typeof (Floor)) .FirstElement() as Floor;
if (floor != null)
{
    FamilySymbolSetIterator symbolItor = family.Symbols.ForwardIterator();
    int x = 0, y = 0;
    int i = 0;
    while (symbolItor.MoveNext())
    {
        FamilySymbol symbol = symbolItor.Current as FamilySymbol;
        XYZ location = new XYZ (x, y, 0);
        XYZ direction = new XYZ();
        switch (i % 3)
        {
            case 0:
                direction = new XYZ (1, 1, 0);
                break;
            case 1:
                direction = new XYZ (0, 1, 1);
                break;
            case 2:
                direction = new XYZ (1, 0, 1);
                break;
        }
        FamilyInstance instance = document.Create.NewFamilyInstance (location, symbol, direction, floor,
        StructuralType.NonStructural);
        x += 10;
        i++;
    }
}
else
{
    throw new Exception ("Please open a model with at least one floor element before invoking this command.");
}
```

代码 3-13 在 Revit 中的结果见图 3-18。

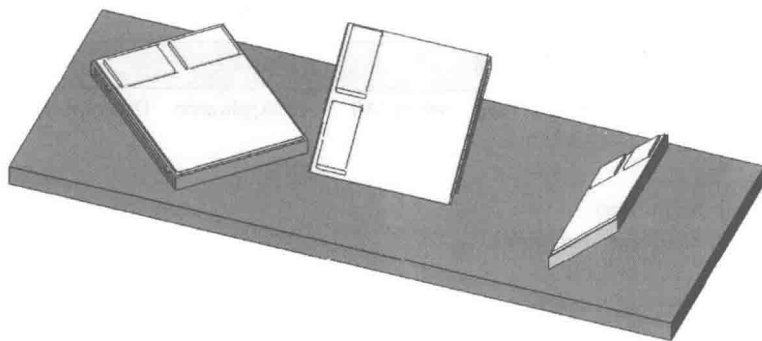


图 3-18 使用参考方向创建族实例



### 3.2.5 族符号 (FamilySymbol)

FamilySymbol 类表示族中单个类型。每个族都可以包含一个或多个族符号。每个族实例有一个与之相关联的族符号，它可以从其 Symbol 属性进行访问。

某些类型的族（门、窗、幕墙面板）包含热属性，如图 3-19 某个窗的类型属性窗口中所示。

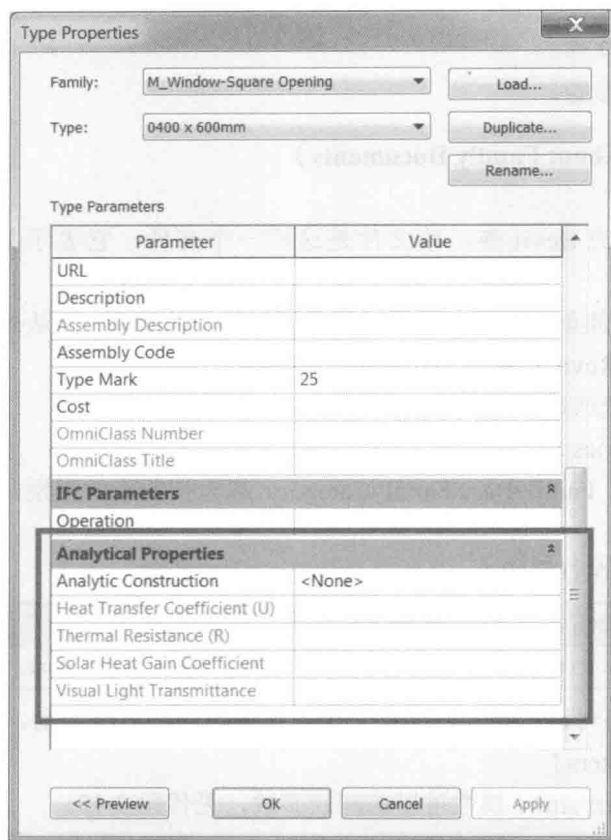


图 3-19 窗的类型属性

族符号的热属性由 FamilyThermalProperties 类表示，用 FamilySymbol.GetThermalProperties() 方法检索。但族符号的 FamilyThermalProperties 可以用 SetThermalProperties() 方法进行设置。FamilyThermalProperties 类的属性本身是只读的。

热属性计算值的单位见表 3-12。

表 3-12

热属性计算值单位

HeatTransferCoefficient (热传导系数)	瓦特每平方米开尔文 [ $\text{W}/(\text{m}^2 \cdot \text{K})$ ]
ThermalResistance (热阻)	平方米开尔文每瓦特 [ $(\text{m}^2 \cdot \text{K})/\text{W}$ ]

AnalyticConstructionTypeId 属性是结构 gbXML (绿色建筑扩展标记语言) 类型，它返回与 Constructions.xml 中 constructionType 节点“ID”属性对应的值。通过 Constructions.xml





中的 `constructionType` 节点“ID”属性，静态方法 `FamilyThermalProperties.Find()` 可找到 `FamilyThermalProperties`。

### 3.3 族文件 (Family Documents)

本节讨论族以及如何：

- 创建和修改族文件。
- 访问族类型和参数。

#### 3.3.1 关于族文件 (About Family Documents)

##### 1. 族 (Family)

族对象表示完整的 Revit 族。族文件是这样文件，它表示某个族而不是 Revit 项目。

使用 Revit API 的族创建功能，可以创建并编辑族及其类型。当从外部系统有现存的可用数据，想要转换为 Revit 族库时，此功能尤其有用。

API 不可以对系统族进行编辑。

##### 2. 类别 (Categories)

正如前一章所述，`FamilyBase.FamilyCategory` 属性指明族的类别，如柱、家具、结构框架或窗。

代码 3-14 可用于在打开的 Revit 族文件中确定族的类别。

#### 代码 3-14: 打开的 Revit 族文件的类别

```
string categoryName = familyDoc.OwnerFamily.FamilyCategory.Name;
```

`FamilyCategory` 还可以被设置，允许一个正在编辑的族更改类别。

##### 3. 参数 (Parameters)

从族文件的 `OwnerFamily` 属性可以访问族参数，见代码 3-15。

#### 代码 3-15: 从族文件访问族参数

```
// get the owner family of the family document.  
Family family = familyDoc.OwnerFamily;  
Parameter param = family.get_Parameter (BuiltInParameter.FAMILY_WORK_PLANE_BASED);  
// this param is a Yes/No parameter in UI, but an integer value in API  
// 1 for true and 0 for false  
int isTrue = param.AsInteger ();  
// param.Set (1); // set value to true.
```

##### 4. 创建族文件 (Creating a Family Document)

如果由 `IsFamilyDocument` 属性判定某文件是族文件，可由 `Document` 类来修改该 Revit 族文件并访问族类型和参数。当使用项目文件时，要编辑已存在的族，可从 `Document` 类中使用 `EditFamily()` 函数，然后在编辑完成后使用 `LoadFamily()` 将族重新载回到主控文件。要新建一个族文件可使用 `Application.NewFamilyDocument()`，见代码 3-16。



### 代码 3-16: 新建族文件

```
// create a new family document using Generic Model.rft template
string templateFileName = @"C: \Documents and Settings\All Users\Application Data\Autodesk\RST 2011\Imperial
Templates\Generic Model.rft";

Document familyDocument = application.NewFamilyDocument (templateFileName);
if (null == familyDocument)
{
    throw new Exception ("Cannot open family document");
}
```

### 5. 内嵌族符号 (Nested Family Symbols)

载入到族的全部族符号可以通过从族文件过滤 FamilySymbols 来获取。代码 3-17 中, 运行结果列出了一个给定族实例的族中所有嵌入的族符号。

### 代码 3-17: 获取族中嵌入的族符号

```
public void GetLoadedSymbols (Autodesk.Revit.DB.Document document, FamilyInstance familyInstance)
{
    if (null != familyInstance.Symbol)
    {
        // Get family associated with this
        Family family = familyInstance.Symbol.Family;

        // Get Family document for family
        Document familyDoc = document.EditFamily (family);
        if (null != familyDoc && familyDoc.IsFamilyDocument == true)
        {
            String loadedFamilies = "FamilySymbols in " + family.Name + ": \n";
            FilteredElementCollector collector = new FilteredElementCollector (document);
            ICollection<Element> collection =
                collector.OfClass (typeof (FamilySymbol)).ToElements();
            foreach (Element e in collection)
            {
                FamilySymbol fs = e as FamilySymbol;
                loadedFamilies += "\t" + fs.Name + "\n";
            }

            TaskDialog.Show ("Revit", loadedFamilies);
        }
    }
}
```

### 3.3.2 在族中创建图元 (Creating Elements in Families)

FamilyItemFactory 类提供了在族文件中创建图元的能力。它通过 Document.Family Create 属性进行访问。FamilyItemFactory 是从 ItemFactoryBase 类派生的, 它是一个在 Revit 项目文件和族文件中创建图元的实用程序。



### 1. 创建形状图元 (Create a Form Element)

FamilyItemFactory 类提供了在族中创建形状图元的能力, 如拉伸、旋转、放样和融合。有关这些三维草图形式的更多信息, 请参阅 3.8.2 节三维草图。

代码 3-18 演示了如何新建拉伸图元。它创建一个简单的矩形轮廓, 然后移动新建的拉伸图元到新位置。

**代码 3-18: 新建拉伸图元**

```
private Extrusion CreateExtrusion (Autodesk.Revit.DB.Document document, SketchPlane sketchPlane)
{
    Extrusion rectExtrusion = null;

    // make sure we have a family document
    if (true == document.IsFamilyDocument)
    {
        // define the profile for the extrusion
        CurveArrArray curveArrArray = new CurveArrArray();
        CurveArray curveArray1 = new CurveArray();
        CurveArray curveArray2 = new CurveArray();
        CurveArray curveArray3 = new CurveArray();

        // create a rectangular profile
        XYZ p0 = XYZ.Zero;
        XYZ p1 = new XYZ (10, 0, 0);
        XYZ p2 = new XYZ (10, 10, 0);
        XYZ p3 = new XYZ (0, 10, 0);
        Line line1 = Line.CreateBound (p0, p1);
        Line line2 = Line.CreateBound (p1, p2);
        Line line3 = Line.CreateBound (p2, p3);
        Line line4 = Line.CreateBound (p3, p0);
        curveArray1.Append (line1);
        curveArray1.Append (line2);
        curveArray1.Append (line3);
        curveArray1.Append (line4);

        curveArrArray.Append (curveArray1);

        // create solid rectangular extrusion
        rectExtrusion = document.FamilyCreate.NewExtrusion (true, curveArrArray, sketchPlane, 10);

        if (null != rectExtrusion)
        {
            // move extrusion to proper place
            XYZ transPoint1 = new XYZ (-16, 0, 0);
            ElementTransformUtils.MoveElement (document, rectExtrusion.Id, transPoint1);
        }
        else
        {
            throw new Exception ("Create new Extrusion failed.");
        }
    }
}
```



```

    }
}
else
{
    throw new Exception ("Please open a Family document before invoking this command.");
}

return rectExtrusion;
}

```

代码 3-19 演示了如何在族文件中从卵形体轮廓创建一个新的放样图元, 创建结果见图 3-20。

#### 代码 3-19: 新建一个放样图元

```

private Sweep CreateSweep (Autodesk.Revit.DB.Document document, SketchPlane sketchPlane)
{
    Sweep sweep = null;

    // make sure we have a family document
    if (true == document.IsFamilyDocument)
    {
        // Define a profile for the sweep
        CurveArrArray arrarr = new CurveArrArray();
        CurveArray arr = new CurveArray();

        // Create an ovoid profile
        XYZ pnt1 = new XYZ (0, 0, 0);
        XYZ pnt2 = new XYZ (2, 0, 0);
        XYZ pnt3 = new XYZ (1, 1, 0);
        arr.Append (Arc.Create (pnt2, 1.0d, 0.0d, 180.0d, XYZ.BasisX, XYZ.BasisY));
        arr.Append (Arc.Create (pnt1, pnt3, pnt2));
        arrarr.Append (arr);
        SweepProfile profile = document.Application.Create.NewCurveLoopsProfile (arrarr);

        // Create a path for the sweep
        XYZ pnt4 = new XYZ (10, 0, 0);
        XYZ pnt5 = new XYZ (0, 10, 0);
        Curve curve = Line.CreateBound (pnt4, pnt5);

        CurveArray curves = new CurveArray();
        curves.Append (curve);

        // create a solid ovoid sweep
        sweep = document.FamilyCreate.NewSweep (true, curves, sketchPlane, profile, 0, ProfilePlaneLocation.Start);

        if (null != sweep)
        {
            // move to proper place
            XYZ transPoint1 = new XYZ (11, 0, 0);

```



```
ElementTransformUtils.MoveElement (document, sweep.Id, transPoint1);
    }
    else
    {
        throw new Exception ("Failed to create a new Sweep.");
    }
}
else
{
    throw new Exception ("Please open a Family document before invoking this command.");
}

return sweep;
}
```



图 3-20 代码 3-19 创建的卵形放样图元

**FreeFormElement** 类允许由输入的体轮廓创建非参数化几何体。**FreeFormElement** 可以参与连接并取消与其他可组合图元的切口。图元平表面可以在面的法线方向以交互方式和编程方式进行拉伸偏移。

在族中新建一个形状之后, 可能需要更改形状的子类别。例如, 项目内有个门族并想创建多个子类别的门, 并指定族中不同的门型以不同的子类别。

代码 3-20 说明了如何新建子类别, 它指定一种材料, 然后将子类别分配给形状图元。

#### 代码 3-20: 指定子类别

```
public void AssignSubCategory (Document document, GenericForm extrusion)
{
    // create a new subcategory
    Category cat = document.OwnerFamily.FamilyCategory;
    Category subCat = document.Settings.Categories.NewSubcategory (cat, "NewSubCat");

    // create a new material and assign it to the subcategory
    ElementId materialId = Material.Create (document, "Wood Material");
    subCat.Material = document.GetElement (materialId) as Material;

    // assign the subcategory to the element
    extrusion.Subcategory = subCat;
}
```

## 2. 创建注释 (Create an Annotation)

在族中还可以新建注释, 诸如尺寸、模型文本和文字注释对象, 以及曲线注释图元如符号曲线、模型曲线及详图曲线。有关注释图元的更多信息, 请参阅 3.6 节注释图元。

此外, 引用一个确定对齐方向的视图, 以及两个几何参照, 还可以添加新的对齐方式。

代码 3-21 演示了如何新建一个弧长尺寸, 创建结果见图 3-21。



### 代码 3-21: 创建尺寸

```
public Dimension CreateArcDimension (Document document, SketchPlane sketchPlane)
{
    Autodesk.Revit.Creation.Application appCreate = document.Application.Create;
    Line gLine1 = Line.CreateBound (new XYZ (0, 2, 0), new XYZ (2, 2, 0));
    Line gLine2 = Line.CreateBound (new XYZ (0, 2, 0), new XYZ (2, 4, 0));
    Arc arcoDim = Arc.Create (new XYZ (1, 2, 0), new XYZ (-1, 2, 0), new XYZ (0, 3, 0));
    Arc arcofDim = Arc.Create (new XYZ (0, 3, 0), new XYZ (1, 2, 0), new XYZ (0.8, 2.8, 0));

    Autodesk.Revit.Creation.FamilyItemFactory creationFamily = document.FamilyCreate;
    ModelCurve modelCurve1 = creationFamily.NewModelCurve (gLine1, sketchPlane);
    ModelCurve modelCurve2 = creationFamily.NewModelCurve (gLine2, sketchPlane);
    ModelCurve modelCurve3 = creationFamily.NewModelCurve (arcoDim, sketchPlane);
    //get their reference
    Reference ref1 = modelCurve1.GeometryCurve.Reference;
    Reference ref2 = modelCurve2.GeometryCurve.Reference;
    Reference arcRef = modelCurve3.GeometryCurve.Reference;

    Dimension newArcDim = creationFamily.NewArcLengthDimension (document.ActiveView, arcofDim, arcRef, ref1, ref2);
    if (newArcDim == null)
    {
        throw new Exception ("Failed to create new arc length dimension.");
    }

    return newArcDim;
}
```

某些类型的尺寸可以用 FamilyParameter 标注。如果试图对无法标注的尺寸获取或设置 Label 属性, 将引发一个非法操作的 Revit 异常。在代码 3-22 中, 在两条线之间新建一个线性尺寸并标注为“宽度”, 创建结果见图 3-22。

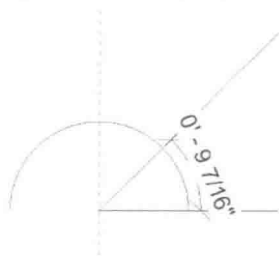


图 3-21 生成弧型长度尺寸

### 代码 3-22: 标注尺寸

```
public Dimension CreateLinearDimension (Document document)
{
    // first create two lines
    XYZ pt1 = new XYZ (5, 5, 0);
    XYZ pt2 = new XYZ (5, 10, 0);

    Line line = Line.CreateBound (pt1, pt2);
    Plane plane = document.Application.Create.NewPlane (pt1.CrossProduct (pt2), pt2);
    SketchPlane skplane = SketchPlane.Create (document, plane);
```



```

ModelCurve modelcurve1 = document.FamilyCreate.NewModelCurve (line, skplane);

pt1 = new XYZ (10, 5, 0);
pt2 = new XYZ (10, 10, 0);
line = Line.CreateBound (pt1, pt2);
plane = document.Application.Create.NewPlane (pt1.CrossProduct (pt2), pt2);
skplane = SketchPlane.Create (document, plane);
ModelCurve modelcurve2 = document.FamilyCreate.NewModelCurve (line, skplane);

// now create a linear dimension between them
ReferenceArray ra = new ReferenceArray( );
ra.Append (modelcurve1.GeometryCurve.Reference);
ra.Append (modelcurve2.GeometryCurve.Reference);

pt1 = new XYZ (5, 10, 0);
pt2 = new XYZ (10, 10, 0);
line = Line.CreateBound (pt1, pt2);

Dimension dim = document.FamilyCreate.NewLinearDimension (document.ActiveView, line, ra);

// create a label for the dimension called "width"
FamilyParameter param = document.FamilyManager.AddParameter ("width",
    BuiltInParameterGroup.PG_CONSTRAINTS,
    ParameterType.Length, false);

dim.FamilyLabel = param;

return dim;
}

```

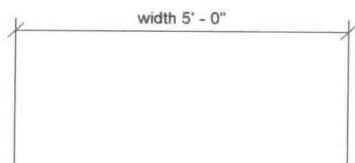


图 3-22 标注线性尺寸

### 3.3.3 族图元的可见性 (Visibility of Family Elements)

FamilyElementVisibility 类可用于控制族图元在项目文件中的可见性。例如, 假定有一个门族, 您可能只希望在该门所在项目文件的平面视图中看到可旋转门, 而在三维视图中不可见。通过设置门旋转曲线的可见性, 可以控制其可见性。FamilyElementVisibility 适用于以下具有 SetVisibility() 方法的族图元类:

- GenericForm, 这是形状图元类的基类, 如放样和拉伸。
- SymbolicCurve。
- ModelText。
- CurveByPoints。
- ModelCurve。
- ReferencePoint。



- ImportInstance。

在代码 3-23 中, 生成的族文件将显示带有下划线的文本“Hello World”。当该族加载到 Revit 项目文件中并放置一个实例时, 在平面视图中, 仅下划线可见。在三维视图中, 下划线和文本都将显示, 但如果过程中设置了详细程度, 则下划线会消失不见。

#### 代码 3-23: 设置族图元的可见性

```
public void CreateAndSetVisibility (Autodesk.Revit.DB.Document familyDocument, SketchPlane sketchPlane)
{
    // create a new ModelCurve in the family document
    XYZ p0 = new XYZ (1, 1, 0);
    XYZ p1 = new XYZ (5, 1, 0);
    Line line1 = Line.CreateBound (p0, p1);

    ModelCurve modelCurve1 = familyDocument.FamilyCreate.NewModelCurve (line1, sketchPlane);

    // create a new ModelText along ModelCurve line
    ModelText text = familyDocument.FamilyCreate.NewModelText ( "Hello World", null, sketchPlane, p0,
HorizontalAlign.Center, 0.1);

    // set visibility for text
    FamilyElementVisibility textVisibility = new FamilyElementVisibility (FamilyElementVisibilityType.Model);
    textVisibility.IsShownInTopBottom = false;
    text.SetVisibility (textVisibility);

    // set visibility for line
    FamilyElementVisibility curveVisibility = new FamilyElementVisibility (FamilyElementVisibilityType.Model);
    curveVisibility.IsShownInCoarse = false;
    modelCurve1.SetVisibility (curveVisibility);
}
```

### 3.3.4 管理族类型和参数 (Managing Family Types and Parameters)

族文件支持对 FamilyManager 属性的访问。FamilyManager 类提供对族类型和参数的访问。使用此类可以添加和删除族类型、族和共享参数, 设置不同族类型的参数值, 并定义公式来驱动参数值。

#### 1. 获取族类型 (Getting Types in a Family)

FamilyManager 可用于迭代族类型, 见代码 3-24, 运行结果见图 3-23。

#### 代码 3-24: 获取族类型

```
public void GetFamilyTypesInFamily (Document familyDoc)
{
    if (familyDoc.IsFamilyDocument == true)
    {
        FamilyManager familyManager = familyDoc.FamilyManager;

        // get types in family
        string types = "Family Types: ";
    }
}
```





```

FamilyTypeSet familyTypes = familyManager.Types;
FamilyTypeSetIterator familyTypesItr = familyTypes.ForwardIterator();
familyTypesItr.Reset();
while (familyTypesItr.MoveNext())
{
    FamilyType familyType = familyTypesItr.Current as FamilyType;
    types += "\n" + familyType.Name;
}
MessageBox.Show (types, "Revit");
}
}

```

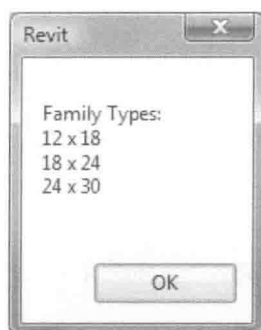


图 3-23 混凝土矩形柱族的族类型

## 2. 编辑族类型 (Editing FamilyTypes)

FamilyManager 可以迭代族的已有类型, 并添加、修改类型及参数。

代码 3-25 演示了如何添加新的类型, 设置其参数, 然后为族实例指定新类型。通过使用 Set() 函数, 可完成对当前类型的类型编辑。从 CurrentType 属性可获得当前类型。CurrentType 属性可用于在编辑前设置当前类型, 或使用 NewType() 函数创建新类型, 并将其设置为当前类型以进行编辑。

注意: 创建和修改新类型之后, 可用 Document.LoadFamily() 将修改后的族载回到 Revit 项目, 以使新类型生效。

### 代码 3-25: 编辑族类型

```

public void EditFamilyTypes (Document document, FamilyInstance familyInstance)
{
    // example works best when familyInstance is a rectangular concrete element
    if (null != familyInstance.Symbol)
    {
        // Get family associated with this
        Family family = familyInstance.Symbol.Family;

        // Get Family document for family
        Document familyDoc = document.EditFamily (family);
        if (null != familyDoc)
        {
            FamilyManager familyManager = familyDoc.FamilyManager;

            // add a new type and edit its parameters
            FamilyType newFamilyType = familyManager.NewType ("2X2");
            // look for 'b' and 'h' parameters and set them to 2 feet
            FamilyParameter familyParam = familyManager.get_Parameter ("b");
            if (null != familyParam)
            {
                familyManager.Set (familyParam, 2.0);
            }
        }
    }
}

```



```

    }
    familyParam = familyManager.get_Parameter ( "h" );
    if ( null != familyParam )
    {
        familyManager.Set ( familyParam, 2.0 );
    }

    // now update the Revit project with Family which has a new type
    family = familyDoc.LoadFamily ( document );
    // find the new type and assign it to FamilyInstance
    FamilySymbolSetIterator symbolsItr = family.Symbols.ForwardIterator ( );
    symbolsItr.Reset ( );
    while ( symbolsItr.MoveNext ( ) )
    {
        FamilySymbol familySymbol = symbolsItr.Current as FamilySymbol;
        if ( familySymbol.Name == "2X2" )
        {
            familyInstance.Symbol = familySymbol;
            break;
        }
    }
}
}
}
}
}

```

### 3.4 概念设计 ( Conceptual Design )

本节论述了 Revit API 中可在族文件中创建复杂几何体的概念设计功能。通过添加新对象可支持“形状生成”：通过点和经过这些点的样条曲线来创建有连续参数关系的可构建形状，其生成几何体的表面可分割、填充图案和镶嵌。

#### 3.4.1 点和曲线对象 ( Point and Curve Objects )

参照点是个用于概念设计环境三维工作空间中指定某个位置的图元。可创建参照点来设计并绘制线、样条线和形状。ReferencePoint 可以添加到 ReferencePointArray 中，然后用于创建 CurveByPoints，也可用于创建形状。

代码 3-26 演示了如何创建 CurveByPoints 对象，创建结果见图 3-24。参阅代码 3-29 了解如何从多个 CurveByPoints 对象创建形状。

##### 代码 3-26：新建“点曲线”

```

ReferencePointArray rpa = new ReferencePointArray ( );

XYZ xyz = document.Application.Create.NewXYZ ( 0, 0, 0 );
ReferencePoint rp = document.FamilyCreate.NewReferencePoint ( xyz );
rpa.Append ( rp );

```



```
xyz = document.Application.Create.NewXYZ (0, 30, 10);  
rp = document.FamilyCreate.NewReferencePoint (xyz);  
rpa.Append (rp);  
  
xyz = document.Application.Create.NewXYZ (0, 60, 0);  
rp = document.FamilyCreate.NewReferencePoint (xyz);  
rpa.Append (rp);  
  
xyz = document.Application.Create.NewXYZ (0, 100, 30);  
rp = document.FamilyCreate.NewReferencePoint (xyz);  
rpa.Append (rp);  
  
xyz = document.Application.Create.NewXYZ (0, 150, 0);  
rp = document.FamilyCreate.NewReferencePoint (xyz);  
rpa.Append (rp);  
  
CurveByPoints curve = document.FamilyCreate.NewCurveByPoints (rpa);
```

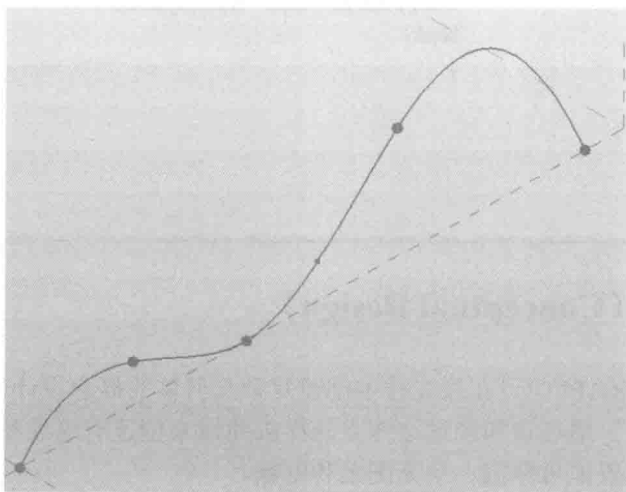


图 3-24 CurveByPoints 曲线

参照点既可以像上例中基于空间坐标创建，也可以相对于其他几何体创建。同时参照几何体改变时，这些参照点也会跟着移动。这些点使用 `PointElementReference` 类的子类来创建。其子类有：

- `PointOnEdge`。
- `PointOnEdgeEdgeIntersection`。
- `PointOnEdgeFaceIntersection`。
- `PointOnFace`。
- `PointOnPlane`。

例如，代码 3-26 中最后两行代码在点曲线的中央创建了一个参照点。

形状可以使用模型线或参照线创建（代码 3-27），模型线会在形状创建期间被形状“消耗”，不再作为独立体存在。而参照线在形状创建后依然存在，并且，如果移动它们，形状



也会更改。尽管 API 未提供 `ReferenceLine` 类, 但可以使用 `ModelCurve.ChangeToReferenceLine()` 方法将模型线转换为参照线。创建结果见图 3-25。

### 代码 3-27: 用参照线创建形状

```
private FormArray CreateRevolveForm (Document document)
{
    FormArray revolveForms = null;

    // Create one profile
    ReferenceArray ref_ar = new ReferenceArray();

    XYZ ptA = new XYZ (0, 0, 10);
    XYZ ptB = new XYZ (100, 0, 10);
    Line line = Line.CreateBound (ptA, ptB);
    ModelCurve modelcurve = MakeLine (document, ptA, ptB);
    ref_ar.Append (modelcurve.GeometryCurve.Reference);

    ptA = new XYZ (100, 0, 10);
    ptB = new XYZ (100, 100, 10);
    modelcurve = MakeLine (document, ptA, ptB);
    ref_ar.Append (modelcurve.GeometryCurve.Reference);

    ptA = new XYZ (100, 100, 10);
    ptB = new XYZ (0, 0, 10);
    modelcurve = MakeLine (document, ptA, ptB);
    ref_ar.Append (modelcurve.GeometryCurve.Reference);

    // Create axis for revolve form
    ptA = new XYZ (-5, 0, 10);
    ptB = new XYZ (-5, 10, 10);
    ModelCurve axis = MakeLine (document, ptA, ptB);

    // make axis a Reference Line
    axis.ChangeToReferenceLine();

    // Typically this operation produces only a single form,
    // but some combinations of arguments will create multiple forms from a single profile.
    revolveForms = document.FamilyCreate.NewRevolveForms (true, ref_ar, axis.GeometryCurve.Reference, 0, Math.PI / 4);

    return revolveForms;
}

public ModelCurve MakeLine (Document doc, XYZ ptA, XYZ ptB)
{
    Autodesk.Revit.ApplicationServices.Application app = doc.Application;
    // Create plane by the points
    Line line = Line.CreateBound (ptA, ptB);
    XYZ norm = ptA.CrossProduct (ptB);
    if (norm.IsZeroLength()) norm = XYZ.BasisZ;
```



```
Plane plane = app.Create.NewPlane (norm, ptB);  
SketchPlane skplane = SketchPlane.Create (doc, plane);  
// Create line here  
ModelCurve modelcurve = doc.FamilyCreate.NewModelCurve (line, skplane);  
return modelcurve;  
}
```

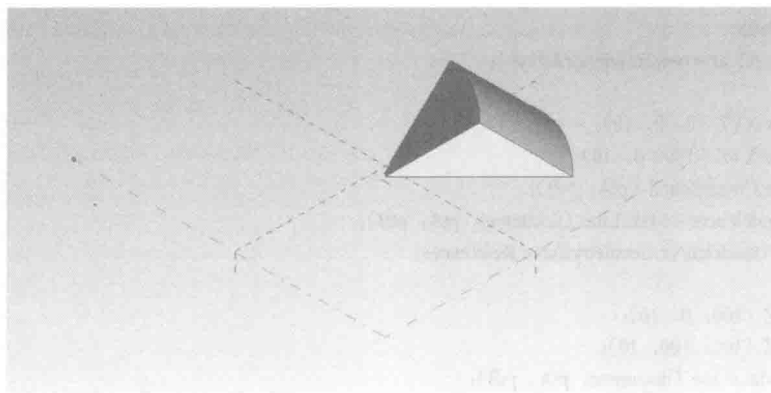


图 3-25 生成旋转形状

### 3.4.2 形状 (Forms)

#### 1. 创建形状 (Creating Forms)

类似于族创建，概念设计环境提供了创建新形状的功能。可以创建以下类型的形状：拉伸、旋转、放样、放样融合、旋转和表面形状。不同于族创建中所使用的 Blend、Extrusion、Revolution、Sweep 和 SweptBlend 类，体量族对所有类型的形状都使用 Form 类。

拉伸形状由闭合的平面曲线环创建。旋转形状由轮廓及同一平面内的一条线来创建，线用于定义轮廓的旋转轴，二维模型轮廓绕该轴旋转后形成三维形状。放样形状由二维轮廓沿着一个平面路径放样来创建。放样融合由多个轮廓创建，沿单一曲线放样每一个平面。融合形状由两个或更多位于不同平面的轮廓创建。单一表面形状由一个轮廓创建，类似于拉伸，但没有给定高度。

代码 3-28 创建了一个简单的拉伸形状。请注意，由于创建形状所用的模型曲线未转换为参考线，因此它们将被生成的形状“吃掉”，见图 3-26。

#### 代码 3-28：创建拉伸形状

```
private Form CreateExtrusionForm (Autodesk.Revit.DB.Document document)  
{  
    Form extrusionForm = null;  
  
    // Create one profile  
    ReferenceArray ref_ar = new ReferenceArray ();  
  
    XYZ ptA = new XYZ (10, 10, 0);  
    XYZ ptB = new XYZ (90, 10, 0);
```



```

ModelCurve modelcurve = MakeLine ( document, ptA, ptB);
ref_ar.Append (modelcurve.GeometryCurve.Reference);

ptA = new XYZ (90, 10, 0);
ptB = new XYZ (10, 90, 0);
modelcurve = MakeLine (document, ptA, ptB);
ref_ar.Append (modelcurve.GeometryCurve.Reference);

ptA = new XYZ (10, 90, 0);
ptB = new XYZ (10, 10, 0);
modelcurve = MakeLine (document, ptA, ptB);
ref_ar.Append (modelcurve.GeometryCurve.Reference);

// The extrusion form direction
XYZ direction = new XYZ (0, 0, 50);

extrusionForm = document.FamilyCreate.NewExtrusionForm (true, ref_ar, direction);

int profileCount = extrusionForm.ProfileCount;

return extrusionForm;
}

public ModelCurve MakeLine (Document doc, XYZ ptA, XYZ ptB)
{
    Autodesk.Revit.ApplicationServices.Application app = doc.Application;
    // Create plane by the points
    Line line = Line.CreateBound (ptA, ptB);
    XYZ norm = ptA.CrossProduct (ptB);
    if (norm.IsZeroLength()) norm = XYZ.BasisZ;
    Plane plane = app.Create.NewPlane (norm, ptB);
    SketchPlane skplane = SketchPlane.Create (doc, plane);
    // Create line here
    ModelCurve modelcurve = doc.FamilyCreate.NewModelCurve (line, skplane);
    return modelcurve;
}

```

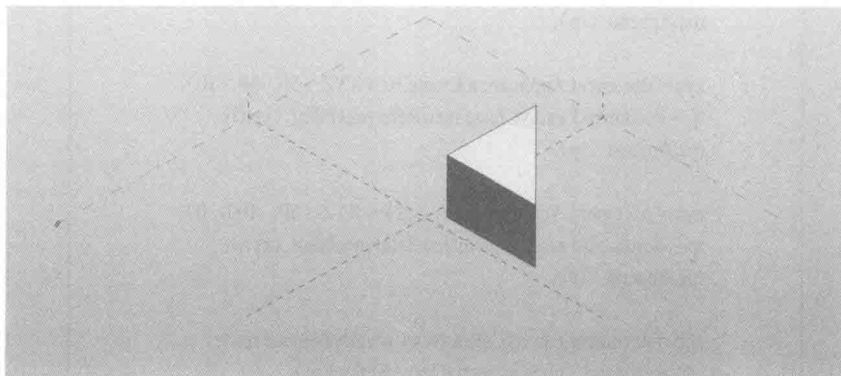


图 3-26 生成拉伸形状



代码 3-29 演示了如何使用一系列 CurveByPoints 对象来创建放样形状, 创建结果见图 3-27。

**代码 3-29: 创建放样形状**

```
private Form CreateLoftForm (Autodesk.Revit.Document document)
{
    Form loftForm = null;

    ReferencePointArray rpa = new ReferencePointArray();
    ReferenceArrayArray ref_ar_ar = new ReferenceArrayArray();
    ReferenceArray ref_ar = new ReferenceArray();
    ReferencePoint rp = null;
    XYZ xyz = null;

    // make first profile curve for loft
    xyz = document.Application.Create.NewXYZ (0, 0, 0);
    rp = document.FamilyCreate.NewReferencePoint (xyz);
    rpa.Append (rp);

    xyz = document.Application.Create.NewXYZ (0, 50, 10);
    rp = document.FamilyCreate.NewReferencePoint (xyz);
    rpa.Append (rp);

    xyz = document.Application.Create.NewXYZ (0, 100, 0);
    rp = document.FamilyCreate.NewReferencePoint (xyz);
    rpa.Append (rp);

    CurveByPoints cbp = document.FamilyCreate.NewCurveByPoints (rpa);
    ref_ar.Append (cbp.GeometryCurve.Reference);
    ref_ar_ar.Append (ref_ar);
    rpa.Clear();
    ref_ar = new ReferenceArray();

    // make second profile curve for loft
    xyz = document.Application.Create.NewXYZ (50, 0, 0);
    rp = document.FamilyCreate.NewReferencePoint (xyz);
    rpa.Append (rp);

    xyz = document.Application.Create.NewXYZ (50, 50, 30);
    rp = document.FamilyCreate.NewReferencePoint (xyz);
    rpa.Append (rp);

    xyz = document.Application.Create.NewXYZ (50, 100, 0);
    rp = document.FamilyCreate.NewReferencePoint (xyz);
    rpa.Append (rp);

    cbp = document.FamilyCreate.NewCurveByPoints (rpa);
    ref_ar.Append (cbp.GeometryCurve.Reference);
    ref_ar_ar.Append (ref_ar);
}
```



```

rpa.Clear();
ref_ar = new ReferenceArray();

// make third profile curve for loft
xyz = document.Application.Create.NewXYZ (75, 0, 0);
rp = document.FamilyCreate.NewReferencePoint (xyz);
rpa.Append (rp);

xyz = document.Application.Create.NewXYZ (75, 50, 5);
rp = document.FamilyCreate.NewReferencePoint (xyz);
rpa.Append (rp);

xyz = document.Application.Create.NewXYZ (75, 100, 0);
rp = document.FamilyCreate.NewReferencePoint (xyz);
rpa.Append (rp);

cbp = document.FamilyCreate.NewCurveByPoints (rpa);
ref_ar.Append (cbp.GeometryCurve.Reference);
ref_ar_ar.Append (ref_ar);

loftForm = document.FamilyCreate.NewLoftForm (true, ref_ar_ar);

return loftForm;
}

```

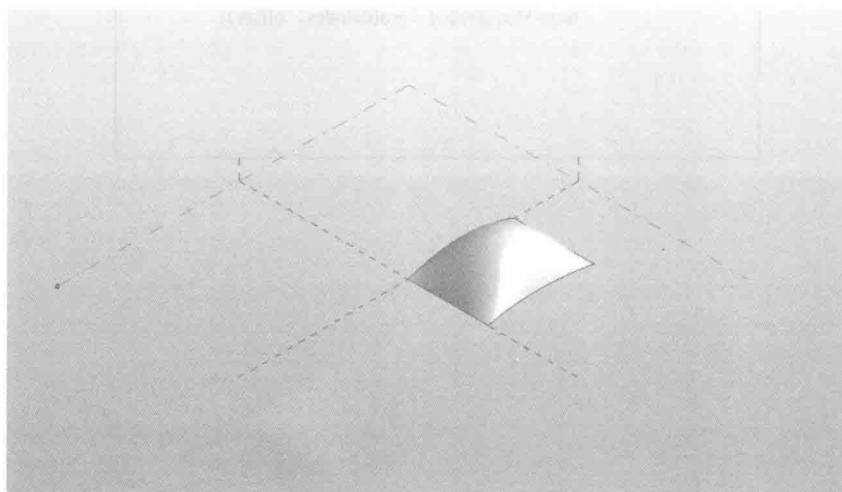


图 3-27 生成放样形状

## 2. 形状修改 (Form Modification)

形状一旦创建后, 可通过改变某个子图元 (如表面、边、曲线或顶点) 或整个轮廓来修改, 修改形状的方法包括:

- AddEdge (添加边)。
- AddProfile (添加轮廓)。





- DeleteProfile (删除轮廓)。
- DeleteSubElement (删除子图元)。
- MoveProfile (移动轮廓)。
- MoveSubElement (移动子图元)。
- RotateProfile (旋转轮廓)。
- RotateSubElement (旋转子图元)。
- ScaleSubElement (缩放子图元)。

此外, 可以通过先添加一条边或一个轮廓来修改形状, 然后再使用上面列出的方法修改。

代码 3-30 按指定的偏移量, 移动形状的第一条轮廓曲线。相应的图 3-28 显示了对上个例子中放样形状应用此代码的结果。

#### 代码 3-30: 移动轮廓

```
public void MoveForm (Form form)
{
    int profileCount = form.ProfileCount;
    if (form.ProfileCount > 0)
    {
        int profileIndex = 0;    // modify the first form only
        if (form.CanManipulateProfile (profileIndex))
        {
            XYZ offset = new XYZ (-25, 0, 0);
            form.MoveProfile (profileIndex, offset);
        }
    }
}
```

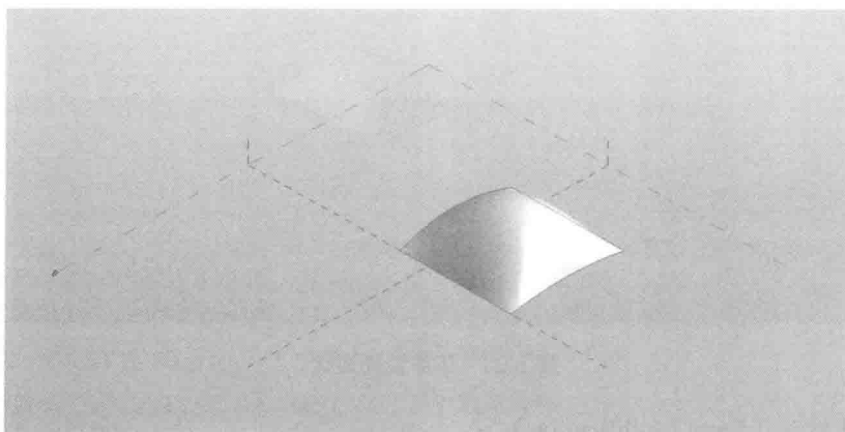


图 3-28 修改后的放样形状

代码 3-31 演示了如何移动给定形状的单个顶点, 图 3-29 显示了此代码应用于先前代码 3-28 创建的拉伸形状示例所产生的影响。



### 代码 3-31: 移动子图元

```
public void MoveSubElement (Form form)
{
    if (form.ProfileCount > 0)
    {
        int profileIndex = 0;    // get first profile
        ReferenceArray ra = form.get_CurveLoopReferencesOnProfile (profileIndex, 0);
        foreach (Reference r in ra)
        {
            ReferenceArray ra2 = form.GetControlPoints (r);
            foreach (Reference r2 in ra2)
            {
                Point vertex = document.GetElement (r2).GetGeometryObjectFromReference (r2) as Point;

                XYZ offset = new XYZ (0, 15, 0);
                form.MoveSubElement (r2, offset);
                break;    // just move the first point
            }
        }
    }
}
```

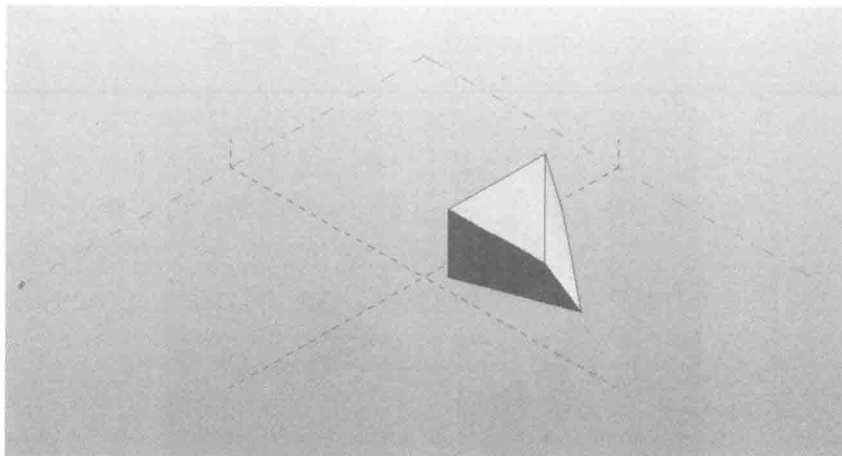


图 3-29 修改后的拉伸形状

### 3.4.3 有理化处理表面 (Rationalizing a Surface)

#### 1. 分割表面 (Dividing a surface)

形状的表面可以用 UV 网格进行分割。使用 `Form.GetDividedSurfaceData()` 方法可以访问表面分割的数据 (代码 3-32、图 3-30), 以及在形状上新建分割表面。

### 代码 3-32: 分割表面

```
public void DivideSurface (Document document, Form form)
{
    Application application = document.Application;
```



```
Options opt = application.Create.NewGeometryOptions();
opt.ComputeReferences = true;

Autodesk.Revit.Geometry.Element geomElem = form.get_Geometry (opt);
foreach (GeometryObject geomObj in geomElem)
{
    Solid solid = geomObj as Solid;
    foreach (Face face in solid.Faces)
    {
        if (face.Reference != null)
        {
            DividedSurface ds = document.FamilyCreate.NewDividedSurface (face.Reference);
            // create a divided surface with fixed number of U and V grid lines
            SpacingRule srU = ds.UspaceRule;
            srU.SetLayoutFixedNumber (16, SpacingRuleJustification.Center, 0, 0);

            SpacingRule srV = ds.VSpaceRule;
            srV.SetLayoutFixedNumber (24, SpacingRuleJustification.Center, 0, 0);

            break;    // just divide one face of form
        }
    }
}
```

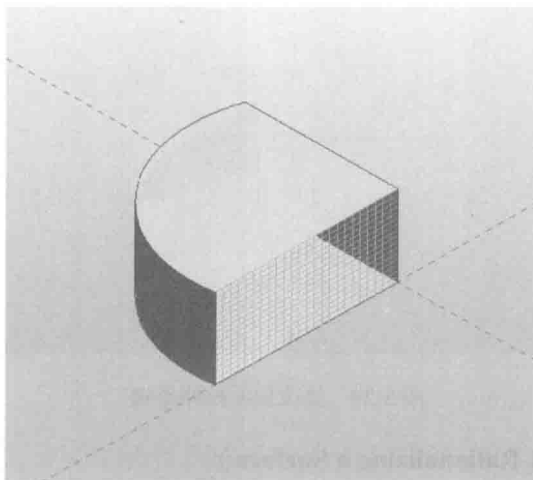


图 3-30 用 UV 网格分割形状表面

访问分割表面的 **USpacing** 和 **VSpacing** 属性, 通过指定确定的网格数 (如上例所示)、确定的网格间距, 或网格间最小或最大间距, 可以定义 **U** 方向和 **V** 方向网格线的 **SpacingRule**。每个间距规则都需要一些附加信息, 如对齐和网格旋转。

## 2. 表面图案填充 (Patterning a Surface)

分割表面可以填充图案。任何内置的瓷砖型式都可应用于分割表面。瓷砖型式是个分



配给 DividedSurface 的图元类型。表面依据 UV 网格布局应用瓷砖型式，因此，改变分割表面的 USpacing 和 VSpacing 属性会影响表面图案外观。

代码 3-33 演示了如何用 OctagonRotate（八角形旋转）图案覆盖分割表面。图 3-31 显示了其应用于先前分割表面示例所产生的外观。请注意，此示例还演示了如何获取形状的分割表面。

**代码 3-33：表面图案填充**

```
public void TileSurface ( Document document, Form form )
{
    // cover surface with OctagonRotate tile pattern
    TilePatterns patterns = document.Settings.TilePatterns;
    DividedSurfaceData dsData = form.GetDividedSurfaceData( );
    if ( dsData != null )
    {
        foreach ( Reference r in dsData.GetReferencesWithDividedSurfaces( ) )
        {
            DividedSurface ds = dsData.GetDividedSurfaceForReference ( r );
            ds.ChangeTypeId ( patterns.GetTilePattern ( TilePatternsBuiltIn.OctagonRotate ) .Id );
        }
    }
}
```

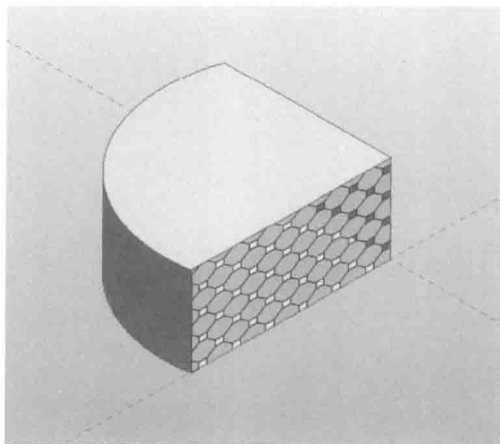


图 3-31 应用于表面的瓷砖型式

除了将内置的瓷砖型式应用于分割表面，还可以使用“Curtain Panel Pattern Based.rft”族样板创建自己的体量嵌板族。通过使用 DividedSurface.ChangeTypeId()方法，这些嵌板族可以载入到体量族并应用于分割表面。

幕墙嵌板族有下列特有族属性：

- IsCurtainPanelFamily（是否为幕墙嵌板族）。
- CurtainPanelHorizontalSpacing（幕墙嵌板水平间距）。
- CurtainPanelVerticalSpacing（幕墙嵌板垂直间距）。
- CurtainPanelTilePattern（幕墙嵌板瓷砖图案）。



代码 3-34 演示了如何编辑体量嵌板族 (图 3-32), 然后将其应用于概念体量文件中的形状 (图 3-33)。运行此示例前, 请先使用 “Curtain Panel Pattern Based.rft” 样板创建一个新的族文件。

#### 代码 3-34: 编辑幕墙嵌板族

```
Family family = document.OwnerFamily;
if (family.IsCurtainPanelFamily == true &&
    family.CurtainPanelTilePattern == TilePatternsBuiltIn.Rectangle)
{
    // first change spacing of grids in family document
    family.CurtainPanelHorizontalSpacing = 20;
    family.CurtainPanelVerticalSpacing = 30;

    // create new points and lines on grid
    Autodesk.Revit.ApplicationServices.Application app = document.Application;
    FilteredElementCollector collector = new FilteredElementCollector (document);
    ICollection<Element> collection = collector.OfClass (typeof (ReferencePoint)).ToElements();
    int ctr = 0;
    ReferencePoint rp0 = null, rp1 = null, rp2 = null, rp3 = null;
    foreach (Autodesk.Revit.DB.Element e in collection)
    {
        ReferencePoint rp = e as ReferencePoint;
        switch (ctr)
        {
            case 0:
                rp0 = rp;
                break;
            case 1:
                rp1 = rp;
                break;
            case 2:
                rp2 = rp;
                break;
            case 3:
                rp3 = rp;
                break;
        }
        ctr++;
    }

    ReferencePointArray rpAr = new ReferencePointArray();
    rpAr.Append (rp0);
    rpAr.Append (rp2);
    CurveByPoints curve1 = document.FamilyCreate.NewCurveByPoints (rpAr);
    PointLocationOnCurve pointLocationOnCurve25 = new PointLocationOnCurve (PointOnCurveMeasurementType.
NormalizedCurveParameter, 0.25, PointOnCurveMeasureFrom.Beginning);
    PointOnEdge poeA = app.Create.NewPointOnEdge (curve1.GeometryCurve.Reference, pointLocationOnCurve25);
    ReferencePoint rpA = document.FamilyCreate.NewReferencePoint (poeA);
    PointLocationOnCurve pointLocationOnCurve75 = new PointLocationOnCurve (PointOnCurve MeasurementType.
NormalizedCurveParameter, 0.75, PointOnCurveMeasureFrom.Beginning);
```



```

PointOnEdge poeB = app.Create.NewPointOnEdge (curve1.GeometryCurve.Reference, pointLocationOnCurve75);
ReferencePoint rpB = document.FamilyCreate.NewReferencePoint (poeB);

rpAr.Clear();
rpAr.Append (rp1);
rpAr.Append (rp3);
CurveByPoints curve2 = document.FamilyCreate.NewCurveByPoints (rpAr);
PointOnEdge poeC = app.Create.NewPointOnEdge (curve2.GeometryCurve.Reference, pointLocationOnCurve25);
ReferencePoint rpC = document.FamilyCreate.NewReferencePoint (poeC);
PointOnEdge poeD = app.Create.NewPointOnEdge (curve2.GeometryCurve.Reference, pointLocationOnCurve75);
ReferencePoint rpD = document.FamilyCreate.NewReferencePoint (poeD);
}
else
{
    throw new Exception ("Please open a curtain family document before calling this command.");
}

```

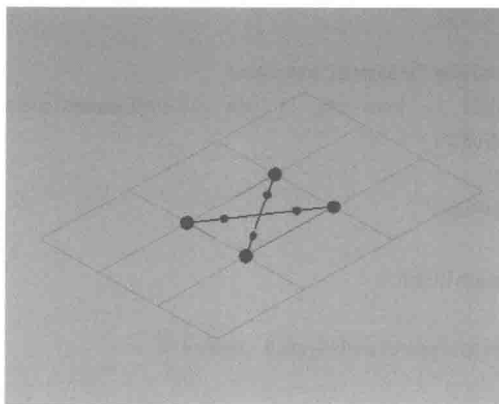


图 3-32 幕墙嵌板族

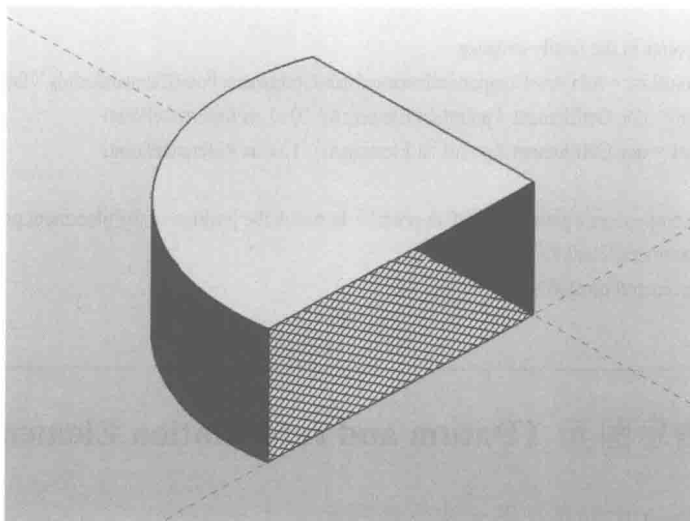


图 3-33 为分割表面指定幕墙嵌板



### 3.4.4 自适应构件 (Adaptive Components)

自适应构件的设计，用于处理构件需要灵活适应许多独特的上下相关联条件的情况。例如，自适应构件可用于根据符合用户定义约束条件布置多个构件而生成的重复系统中。

代码 3-35 演示了如何在体量族中创建自适应构件族实例，并将该实例中的自适应点与体量族中的点对齐。

代码 3-35: 创建自适应构件族实例

```
// find the first placement point that will host the adaptive component
IEnumerable<ReferencePoint> points0 = from obj in new FilteredElementCollector ( doc ) .OfClass ( typeof
(ReferencePoint)) .Cast<ReferencePoint>()
    let type = obj as ReferencePoint
    where type.Name == "PlacementPoint0" // these names were manually assigned to the points
    select obj;
ReferencePoint placementPoint0 = points0.First();

// find the 2nd placement point that will host the adaptive component
IEnumerable<ReferencePoint> points1 = from obj in new FilteredElementCollector ( doc ) .OfClass ( typeof
(ReferencePoint)) .Cast<ReferencePoint>()
    let type = obj as ReferencePoint
    where type.Name == "PlacementPoint1"
    select obj;
ReferencePoint placementPoint1 = points1.First();

if ( AdaptiveComponentInstanceUtils.IsAdaptiveFamilySymbol (symbol))
{
    // create an instance of the adaptive component
    FamilyInstance familyInstance = AdaptiveComponentInstanceUtils.CreateAdaptiveComponentInstance (doc, symbol);

    // find the adaptive points in the family instance
    IList<ElementId> pointList = AdaptiveComponentInstanceUtils.GetInstancePointElementRefIds (familyInstance);
    ReferencePoint point0 = doc.GetElement (pointList.ElementAt (0)) as ReferencePoint;
    ReferencePoint point1 = doc.GetElement (pointList.ElementAt (1)) as ReferencePoint;

    // move the adaptive component's points (point0 & point1) to match the position of the placement points
    point0.Position = placementPoint0.Position;
    point1.Position = placementPoint1.Position;
}
```

## 3.5 基准和信息图元 (Datum and Information Elements)

本节介绍了 Revit 中的基准图元和信息图元。

- 基准图元包括标高、轴网和模型曲线。



- 信息图元包括阶段、设计选项和能量数据设置。

关于 Revit 图元分类的详细信息，请参阅 1.5 节图元概要。

请注意，更多信息，可参考相关章节：

- LoadBase、LoadCase、LoadCombination、LoadNature 及 LoadUsage，参考 4.2 节 Revit Structure。
- 模型曲线，参考 3.8 节草图。
- 材料及填充图案，参考 3.9 节材料。
- 能量数据设置，参考 5.11.1 节能量数据。

### 3.5.1 标高 (Levels)

标高是有限的水平平面，作为“标高托管”图元（如墙、屋顶、楼板和天花板）的参照。在 Revit 平台 API 中，Level 类由 Element 类派生。继承的 Name 属性用于检索 Revit 用户界面中的标高符号旁的用户可见的标高名称。要检索项目中所有的标高，使用 ElementIterator 迭代器来搜索 Level 对象。

#### 1. 高程 (Elevation)

Level 类具有以下属性：

- Elevation 属性 (LEVEL\_ELEV) 用于检索或更改高于或低于地面标高的高程，见图 3-34。
- ProjectElevation 属性用于检索与 Elevation Base（基面）参数值无关、相对于项目原点的标高。
- Elevation Base 是一个 Level 类型的参数。
  - 其内建参数是 LEVEL\_RELATIVE\_BASE\_TYPE。
  - 其存储类型是整型。
  - 0 对应于项目，1 对应于共享。

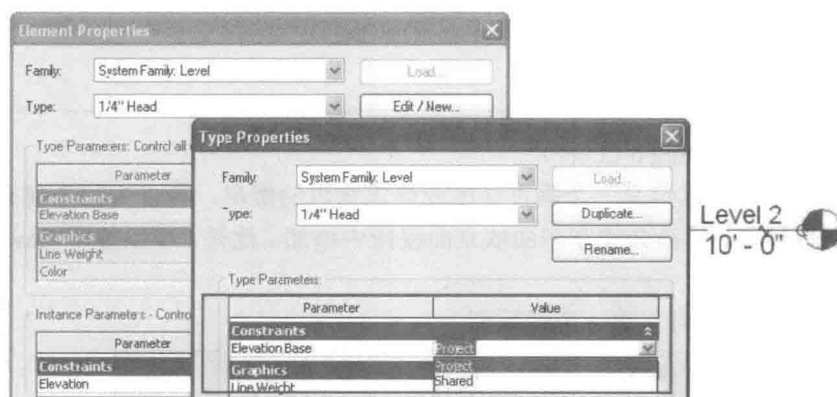


图 3-34 Level 类型 Elevation Base 属性

代码 3-36 说明了如何使用 ElementIterator 检索项目中所有的标高。



**代码 3-36: 检索所有标高**

```
private void Getinfo_Level (Document document)
{
    StringBuilder levelInformation = new StringBuilder();
    int levelNumber = 0;
    FilteredElementCollector collector = new FilteredElementCollector (document);
    ICollection<Element> collection = collector.OfClass (typeof (Level)).ToElements();
    foreach (Element e in collection)
    {
        Level level = e as Level;

        if (null != level)
        {
            // keep track of number of levels
            levelNumber++;

            //get the name of the level
            levelInformation.Append ("\\nLevel Name: " + level.Name);

            //get the elevation of the level
            levelInformation.Append ("\\n\\tElevation: " + level.Elevation);

            // get the project elevation of the level
            levelInformation.Append ("\\n\\tProject Elevation: " + level.ProjectElevation);
        }
    }

    //number of total levels in current document
    levelInformation.Append ("\\n\\n There are " + levelNumber + " levels in the document!");

    //show the level information in the messagebox
    TaskDialog.Show ("Revit", levelInformation.ToString());
}
```

**2. 创建标高 (Creating a Level)**

使用标高命令, 可以定义一个垂直高度或建筑物内的楼层, 可以为每个现有楼层或其他建筑参照创建标高。标高必须在剖面或立面视图中添加。此外, 可以使用 Revit 平台 API 创建新的标高。

代码 3-37 演示了如何创建一个新标高。

**代码 3-37: 创建新标高**

```
Level CreateLevel (Autodesk.Revit.Document document)
{
    // The elevation to apply to the new level
    double elevation = 20.0;

    // Begin to create a level
```



```

Level level = document.Create.NewLevel (elevation);
if (null == level)
{
    throw new Exception ("Create a new level failed.");
}

// Change the level name
level.Name = "New level";

return level;
}

```

注：在创建新标高后，Revit 不会为此标高创建相关联的平面视图，如果需要，用户可以自己创建。有关如何创建平面视图的更多信息，请参考视图。

### 3.5.2 轴网 (Grids)

轴网由 Element 类派生的 Grid 类来表示。它包含所有的轴网属性和方法。继承的 Name 属性用于检索轴网线的编号圈内容。

#### 1. 曲线 (Curve)

Grid 类 Curve 属性获取一个对象，该对象代表轴线几何形状。

- 如果 IsCurved 属性返回 true，则 Curve 属性为 Arc 类对象。
- 如果 IsCurved 属性返回 false，则 Curve 属性为 Line 类对象。

代码 3-38 是一个使用 Grid 类的简单示例。在调用命令后，结果会显示在消息框中。

#### 代码 3-38：使用 Grid 类

```

public void GetInfo_Grid (Grid grid)
{
    string message = "Grid : ";

    // Show IsCurved property
    message += "\nIf grid is Arc : " + grid.IsCurved;

    // Show Curve information
    Autodesk.Revit.DB.Curve curve = grid.Curve;
    if (grid.IsCurved)
    {
        // if the curve is an arc, give center and radius information
        Autodesk.Revit.DB.Arc arc = curve as Autodesk.Revit.DB.Arc;
        message += "\nArc's radius: " + arc.Radius;
        message += "\nArc's center: (" + XYZToString (arc.Center);
    }
    else
    {
        // if the curve is a line, give length information
        Autodesk.Revit.DB.Line line = curve as Autodesk.Revit.DB.Line;
        message += "\nLine's Length: " + line.Length;
    }
}

```



```
}  
// Get curve start point  
message += "\nStart point: " + XYZToString (curve.GetEndPoint (0));  
// Get curve end point  
message += "; End point: " + XYZToString (curve.GetEndPoint (0));  
  
TaskDialog.Show ("Revit", message);  
}  
  
// output the point's three coordinates  
string XYZToString (XYZ point)  
{  
    return "(" + point.X + ", " + point.Y + ", " + point.Z + ")";  
}
```

## 2. 创建轴网 (Creating a Grid)

在 Revit 平台 API 中, 有两个重载 Document 方法可用于创建新的轴网。采用不同的参数使用代码 3-39 方法, 可以创建曲线或直线轴网。

### 代码 3-39: NewGrid()

```
public Grid NewGrid (Arc arc);  
public Grid NewGrid (Line line);
```

注意: 用于创建轴网的弧线或直线必须在水平面内。

代码 3-40 演示了如何用直线或弧线新建轴网。

### 代码 3-40: 用直线或弧线创建轴网

```
void CreateGrid (Autodesk.Revit.DB.Document document)  
{  
    // Create the geometry line which the grid locates  
    XYZ start = new XYZ (0, 0, 0);  
    XYZ end = new XYZ (30, 30, 0);  
    Line geomLine = Line.CreateBound (start, end);  
  
    // Create a grid using the geometry line  
    Grid lineGrid = document.Create.NewGrid (geomLine);  
  
    if (null == lineGrid)  
    {  
        throw new Exception ("Create a new straight grid failed.");  
    }  
  
    // Modify the name of the created grid  
    lineGrid.Name = "New Name1";  
  
    // Create the geometry arc which the grid locates  
    XYZ end0 = new XYZ (0, 0, 0);  
    XYZ end1 = new XYZ (10, 40, 0);
```



```
XYZ pointOnCurve = new XYZ (5, 7, 0);
Arc geomArc = Arc.Create (end0, end1, pointOnCurve);

// Create a grid using the geometry arc
Grid arcGrid = document.Create.NewGrid (geomArc);

if (null == arcGrid)
{
    throw new Exception ("Create a new curved grid failed.");
}

// Modify the name of the created grid
arcGrid.Name = "New Name2";
}
```

注意：在 Revit 中，轴网创建时会以数字或字母顺序自动命名。

多个轴网可以使用 `Document.NewGrids()` 方法同时进行创建，该方法采用 `CurveArray` 参数。

### 3.5.3 阶段 (Phase)

一些建筑项目需要分阶段进行，如翻修。阶段具有以下特点：

- 阶段代表项目生命周期的不同时间段。
- 建筑物内的图元生存期由阶段控制。
- 每个图元都有施工阶段的属性，但只有有限生存期的图元具有拆除阶段的属性。

项目的所有阶段都可以从 `Document` 对象中检索。`Phase` 对象包含三条有用的信息：`Name`、`ID` 和 `UniqueId`。其余属性总是返回 `null` 或空集合。

每个新添加到项目的模型构件都会有一个 `Created Phase ID`（创建阶段 ID）和 `Demolished Phase ID`（拆除阶段 ID）属性。`Element.AllowPhases()` 方法指示其阶段 ID 属性是否可以修改。

创建阶段 ID 属性具有以下特点：

- 识别构件被添加于何阶段。
- 默认值为与当前视图 `Phase` 属性相同的 ID。
- 通过选择相应的下拉列表中的新值，可以更改 `Created Phase ID` 参数。

拆除阶段 ID 属性具有以下特点：

- 识别构件被拆除于何阶段。
- 默认值为 `none`。
- 用拆除工具拆除构件，会在拆除图元的视图中更新当前 `Phase ID` 属性值。
- 通过设置 `Demolished Phase ID` 属性为不同的值，可以拆除某个构件。
- 如果使用 Revit 平台 API 删除某个阶段，当前阶段的所有模型构件仍然存在。这些构件的 `Created Phase ID` 参数值更改为属性对话框下拉列表中的下一项，见图 3-35。

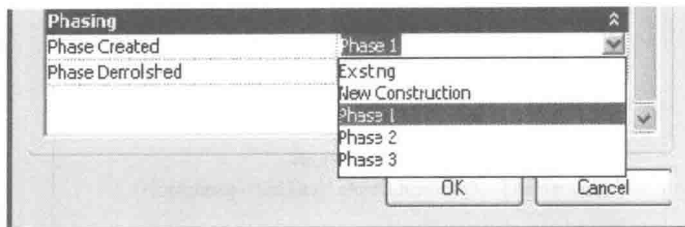


图 3-35 创建阶段构件参数值

代码 3-41 显示了当前文件中所支持的全部阶段。阶段名称显示在消息框中。

**代码 3-41：显示受支持的所有阶段**

```
void Getinfo_Phase (Document doc)
{
    // Get the phase array which contains all the phases.
    PhaseArray phases = doc.Phases;
    // Format the string which identifies all supported phases in the current document.
    String prompt = null;
    if (0 != phases.Size)
    {
        prompt = "All the phases in current document list as follow: ";
        foreach (Phase ii in phases)
        {
            prompt += "\n\t" + ii.Name;
        }
    }
    else
    {
        prompt = "There are no phases in current document.";
    }
    // Give the user the information.
    TaskDialog.Show ("Revit", prompt);
}
```

### 3.5.4 设计选项 (Design Options)

设计选项提供一种方法来探索项目的备选设计。设计选项能灵活适应项目范围中的变更或提供可供评审的备选设计。从使用项目的主模型开始，随项目推进向用户展示不同的设计。大多数图元都可以添加到设计选项中。但不要将主模型中经过深思熟虑的以及没有可替代设计方案的那部分图元添加到设计选项中。

设计选项主要用作 `Element` 类的属性，请参见代码 3-42。

**代码 3-42：使用设计选项**

```
void Getinfo_DesignOption (Document document)
{
    // Get the selected Elements in the Active Document
    UIDocument uidoc = new UIDocument (document);
    ElementSet selection = uidoc.Selection.Elements;
```



```
foreach (Autodesk.Revit.DB.Element element in selection)
{
    // Use the DesignOption property of Element
    if (element.DesignOption != null)
    {
        TaskDialog.Show ("Revit", element.DesignOption.Name.ToString());
    }
}
}
```

下列规则适用于设计选项：

- 如果是主模型中的图元，则 `DesignOption` 属性值为 `null`。否则，返回该图元在 Revit UI 中创建的名称。
- 活动文件中只能有一个活动的设计选项图元。主选项是默认的活动设计选项。例如，名为墙的设计选项集中，有两个名为“砖墙”和“玻璃墙”的设计选项，如果“砖墙”是主选项，则只有此选项和属于它的图元可由 `Element` 过滤器检索到，而“玻璃墙”处于非活动状态。

## 3.6 注释图元 (Annotation Elements)

本节介绍了 Revit 注释图元，包括以下内容：

- `Dimension` (尺寸)。
- `DetailCurve` (详图曲线)。
- `IndependentTag` (独立标记)。
- `TextNote` (文字注释)。
- `AnnotationSymbol` (注释符号)。

注意：

- 尺寸是视图专有图元，在项目中它表示尺寸和距离。
- 详图曲线为详图图纸而创建，仅在其被绘制的视图中可见。它们通常在模型视图中绘制。
- 标记是用于识别图纸中图元的注释，与标记关联的属性会出现在明细表中。
- 注释符号载入项目时，可有多个引线选项。
- 关于 Revit 图元分类的详细信息，请参阅 1.5 节图元概要。

### 3.6.1 尺寸和限制条件 (Dimensions and Constraints)

`Dimension` 类表示永久性尺寸和相关限制条件图元尺寸。用户界面中编辑图元产生的临时尺寸是无法访问的。而高程点和高程点坐标是由 `SpotDimension` 类来表示的。

代码 3-43 的尾段阐释了如何由限制条件图元辨别永久性尺寸。

**代码 3-43：由限制条件辨别永久性尺寸**

```
public void GetInfo_Dimension (Dimension dimension)
{
```



```
string message = "Dimension : ";
// Get Dimension name
message += "\nDimension name is : " + dimension.Name;

// Get Dimension Curve
Autodesk.Revit.DB.Curve curve = dimension.Curve;
if (curve != null && curve.IsBound)
{
    // Get curve start point
    message += "\nCurve start point: (" + curve.GetEndPoint (0) .X + ", "
        + curve.GetEndPoint (0) .Y + ", " + curve.GetEndPoint (0) .Z + ") ";
    // Get curve end point
    message += "; Curve end point: (" + curve.GetEndPoint (1) .X + ", "
        + curve.GetEndPoint (1) .Y + ", " + curve.GetEndPoint (1) .Z + ") ";
}

// Get Dimension type name
message += "\nDimension type name is : " + dimension.DimensionType.Name;

// Get Dimension view name
message += "\nDimension view name is : " + dimension.View.Name;

// Get Dimension reference count
message += "\nDimension references count is " + dimension.References.Size;

if ((int) BuiltInCategory.OST_Dimensions == dimension.Category.Id.IntegerValue)
{
    message += "\nDimension is a permanent dimension.";
}
else if ((int) BuiltInCategory.OST_Constraints == dimension.Category.Id.IntegerValue)
{
    message += "\nDimension is a constraint element.";
}

TaskDialog.Show ("Revit", message);
}
```

### 1. 尺寸 (Dimensions)

五种永久性尺寸 (图 3-36) 如下:

- Linear (线性)。
- Radial (半径)。
- Diameter (直径)。
- Angular (角度)。
- Arc length (弧长)。

所有永久性尺寸的内建类别都是 OST\_Dimensions。API 没有简单的方法来辨别以上五种尺寸。

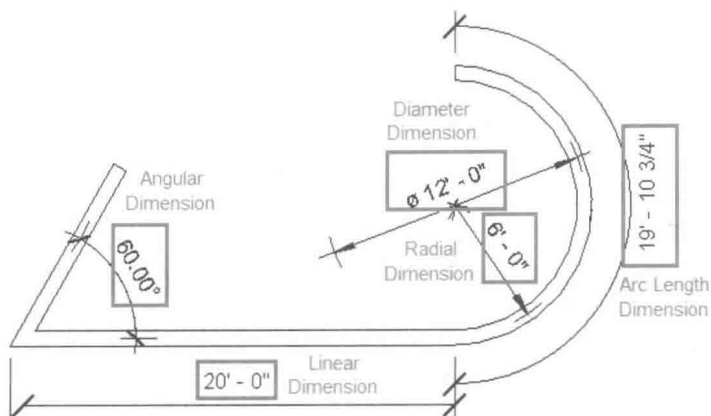


图 3-36 永久性尺寸

除半径和直径尺寸外，每个尺寸都有个尺寸线。尺寸线可从 `Dimension.Curve` 属性获取，该属性始终是未绑定的。换句话说，尺寸线没有起点或终点。基于上图：

- 线性尺寸返回一个直线对象。
- 弧长或角度尺寸返回一个弧对象。
- 半径尺寸返回 `null`。
- 直径尺寸返回 `null`。

通过选取几何参照，如图 3-37 所示创建尺寸。在 API 中，几何参照表示为 `Reference` 类。图 3-38 中尺寸参照可从 `Reference` 属性获得。有关参照的更多信息，请参阅 3.7.2 节中的参照。

- 半径和直径：返回一个曲线参照对象。
- 角度和弧长：返回两个参照对象。
- 线性尺寸：返回两个参照对象。图 3-38 中，线性尺寸有五个参照对象。

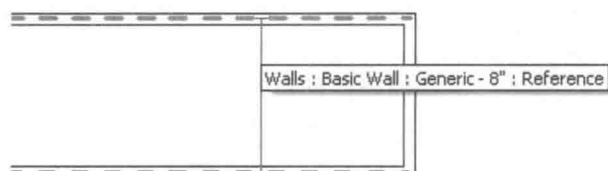


图 3-37 尺寸参照

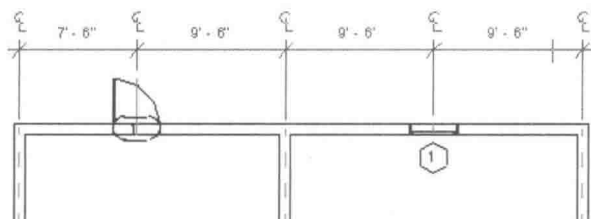


图 3-38 线性尺寸参照

像其他注释图元一样，尺寸为视图专有图元，只在添加它们的视图中显示。Dimension.





View 属性返回该特定视图。

## 2. 限制条件图元 (Constraint Elements)

有限制条件类别 (BuiltInCategory.OST\_Constraints) 的尺寸对象代表两类尺寸相关的限制条件:

- 线性和半径尺寸限制条件。
- 相等限制条件。

在图 3-39 中, 两种锁定的限制条件与线性和半径尺寸对应。在应用程序中, 它们显示为带绿色虚线的挂锁 (绿色虚线可从 Dimension.Curve 属性获取)。线性和半径尺寸限制条件都会由 Dimension.References 属性返回两个 Reference 对象。

限制条件图元为非视图专有图元, 可在不同的视图中显示。因此, View 属性总是返回 null。图 3-39 中的限制条件图元在三维视图也可见, 见图 3-40。

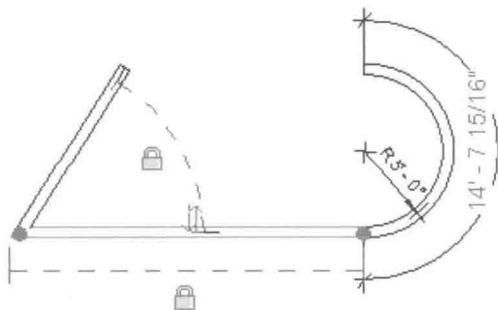


图 3-39 线性和半径尺寸限制条件

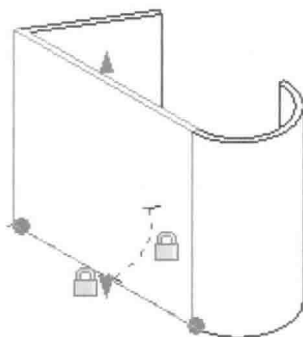


图 3-40 三维视图中的线性和半径尺寸限制条件

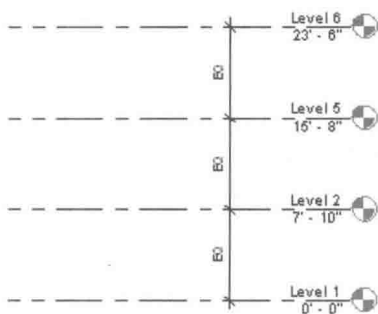


图 3-41 相等限制条件

虽然相等限制条件 (图 3-41) 基于尺寸, 但也可由 Dimension 类来表示。但在 API 中使用类别或 DimensionType, 没有直接的方式从相等限制条件中辨别出线性尺寸限制条件。相等限制条件返回三个或更多参照而线性尺寸限制条件返回两个或多个参照。

注意: 并非所有的限制条件图元都是由 Dimension 类表示, 但它们都属于 Constraints (OST\_Constraints) 类别, 如对齐限制条件。

## 3. 高程点尺寸 (Spot Dimensions)

高程点坐标和高程点 (图 3-42) 由 SpotDimension 类表示并按类别进行区分。和永久性尺寸一样, 高程点尺寸也是视图专有图元。高程点尺寸的类型和类别见表 3-13。

表 3-13

高程点尺寸类型和类别

类 型	类 别
高程点坐标	OST_SpotCoordinates
高程点	OST_SpotElevations



SpotDimension Location 可以向下转换到 LocationPoint, 以便可从 Location Point. Point 属性获得由高程点尺寸表示的高程点坐标。

- 高程点尺寸没有尺寸曲线, 所以它们的 Curve 属性总是返回 null。
- 高程点尺寸 References 属性返回一个参照, 此参照表示高程点尺寸所参照的点或边。
- 要控制文本和标记显示样式, 请修改 SpotDimension 和 SpotDimensionType 参数。

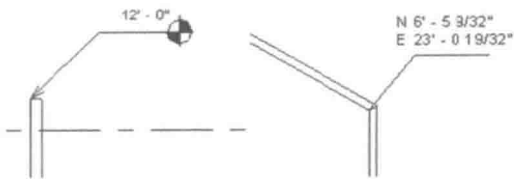


图 3-42 高程点坐标和高程点

4. 比较 (Comparison)

表 3-14 比较了 API 中各种不同的尺寸和限制条件。

表 3-14 尺寸类别比较

尺寸或限制条件	尺寸或限制条件	API 类	内建类别	曲线	参照	视图	位置
永久性尺寸	线性尺寸	Dimension	OST_Dimensions	直线	2	视图专有	null
永久性尺寸	半径尺寸	Dimension	OST_Dimensions	null	1	视图专有	null
永久性尺寸	直径尺寸	Dimension	OST_Dimensions	null	1	视图专有	null
永久性尺寸	角度尺寸	Dimension	OST_Dimensions	弧	2	视图专有	null
永久性尺寸	弧长尺寸	Dimension	OST_Dimensions	弧	2	视图专有	null
尺寸限制条件	线性尺寸限制条件	Dimension	OST_Constraints	弧	2		null
尺寸限制条件	角度尺寸	Dimension	OST_Constraints	弧	2		null
相等限制条件	相等限制条件	Dimension	OST_Constraints	直线	3		null

5. 创建和删除 (Create and Delete)

NewDimension()是 Creation.Document 类中的可用方法 (代码 3-44), 然而此方法仅可创建线性尺寸。

代码 3-44: NewDimension()

```
public Dimension NewDimension (View view, Line line, ReferenceArray references)
public Dimension NewDimension (View view, Line line, ReferenceArray references,
DimensionType dimensionType)
```

使用 NewDimension()方法输入参数, 可定义可见视图、尺寸线以及参照 (两个或更多)。然而, 没有简单的方式来从其他类型中辨别出线性尺寸的尺寸类型。带有尺寸类型参数的重载 NewDimension()方法很少使用。

代码 3-45 说明了如何使用 NewDimension()方法来复制尺寸。

代码 3-45: 用 NewDimension()方法来复制尺寸

```
public void DuplicateDimension (Document document, Dimension dimension)
{
    Line line = dimension.Curve as Line;
    if (null != line)
```



```
{
    Autodesk.Revit.DB.View view = dimension.View;
    ReferenceArray references = dimension.References;
    Dimension newDimension = document.Create.NewDimension (view, line, references);
}
}
```

虽然仅能创建线性尺寸,但 Document.Delete()方法可删除由 Dimension 和 SpotDimension 表示的所有尺寸和限制条件。

### 3.6.2 详图曲线 (Detail Curve)

详图曲线通常是在详图或绘图视图中使用的重要详图构件。详图曲线可从 DetailCurve 类及其派生类中访问 (代码 3-46)。

详图曲线像其他注释图元一样为视图专有,但并无 DetailCurve.View 属性。创建一个详细曲线时,必须将详细曲线与模型曲线视图进行比较。

代码 3-46: NewDetailCurve() and NewModelCurve()

```
public DetailCurve NewDetailCurve (View, Curve, SketchPlane)

public ModelCurve NewModelCurve (Curve, SketchPlane)
```

通常只能在二维视图如标高视图和立面视图中创建详图曲线,否则将引发异常。

除与视图相关的特性不同外,详图曲线与模型曲线非常相似。模型曲线属性和用法的更多信息,请参阅 3.8.3 节模型曲线。

### 3.6.3 标记 (Tags)

标记是用于识别绘图图元的注释。API 公开了独立标记 IndependentTag 和房间标记 RoomTag 类来涵盖大部分 Revit 应用程序中使用的标记。RoomTag 的详细信息,请参阅 4.1 节 Revit Architecture 中的房间。

注意: IndependentTag 类表示 Revit 标记图元和其他专用标记,如注释记号、梁系统标记、电子线路符号 (Revit MEP),等等。在 Revit 内码中,专有标记有从 IndependentTag 派生的相应类。因此,API 未公开其具体特性,且不能用 NewTag()方法创建,它们可按表 3-15 中的类别区分。

表 3-15 标记名称和类别

标记名称	内建类别
Keynote Tag	OST_KeynoteTags
Beam System Tag	OST_BeamSystemTags
Electronic Circuit Tag	OST_ElectricalCircuitTags
Span Direction Tag	OST_SpanDirectionSymbol
Path Reinforcement Span Tag	OST_PathReinSpanSymbol
Rebar System Span Tag	OST_IOSRebarSystemSpanSymbolCtrl



本节中，主要关注图 3-43 所表示的标记类型。

族库中的每个类别都有一个现成的标记。其中一些标记会随默认的 Revit 应用程序样板自动载入，而其他的则需手动加载。如果主体图元是使用 By Category 选项创建的，则 IndependentTag 对象依据主体图元返回不同的类别。例如，墙和楼板 IndependentTag 类别分别是 OST\_WallTags 和 OST\_FloorTags。

如果标记是使用 Multi-Category 或 Material 样式创建的，那么它们的类别分别是 OST\_Multi CategoryTags 和 OST\_MaterialTags。

注意：

- NewTag() 只能在二维视图或锁定的三维视图中使用，否则将引发异常。代码 3-47 说明了如何创建独立标记。在标高视图为活动视图时运行该程序。
- 更改独立标记中显示的文本时，不能直接更改，需要通过更改所标记图元族类型中输入标记文本的参数。代码 3-47 中，参数是“Type Mark”，在 Revit 用户界面族编辑器中也可更改此设置，创建结果见图 3-44。

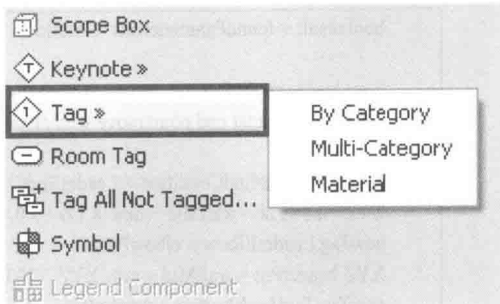


图 3-43 IndependentTag 类

#### 代码 3-47：创建独立标记

```
private IndependentTag CreateIndependentTag (Autodesk.Revit.DB.Document document, Wall wall)
{
    // make sure active view is not a 3D view
    Autodesk.Revit.DB.View view = document.ActiveView;

    // define tag mode and tag orientation for new tag
    TagMode tagMode = TagMode.TM_ADDBY_CATEGORY;
    TagOrientation tagorn = TagOrientation.Horizontal;

    // Add the tag to the middle of the wall
    LocationCurve wallLoc = wall.Location as LocationCurve;
    XYZ wallStart = wallLoc.Curve.GetEndPoint (0);
    XYZ wallEnd = wallLoc.Curve.GetEndPoint (1);
    XYZ wallMid = wallLoc.Curve.Evaluate (0.5, true);

    IndependentTag newTag = document.Create.NewTag (view, wall, true, tagMode, tagorn, wallMid);
    if (null == newTag)
    {
        throw new Exception ("Create IndependentTag Failed.");
    }

    // newTag.TagText is read-only, so we change the Type Mark type parameter to
    // set the tag text. The label parameter for the tag family determines
    // what type parameter is used for the tag text.

    WallType type = wall.WallType;

    Parameter foundParameter = type.get_Parameter ("Type Mark");
```



```
bool result = foundParameter.Set ("Hello");

// set leader mode free
// otherwise leader end point move with elbow point

newTag.LeaderEndCondition = LeaderEndCondition.Free;
XYZ elbowPnt = wallMid + new XYZ (5.0, 5.0, 0.0);
newTag.LeaderElbow = elbowPnt;
XYZ headerPnt = wallMid + new XYZ (10.0, 10.0, 0.0);
newTag.TagHeadPosition = headerPnt;

return newTag;
}
```

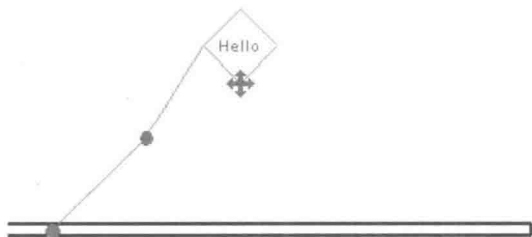


图 3-44 创建独立标记

### 3.6.4 文本 (Text)

在 API 中, TextNote 类表示文本, 其特性概述于表 3-16。

表 3-16 TextNote 特性概述

功 能	API 方法或属性
为应用程序添加文本	Document.Create.NewTextNote( )方法
由文本组件获取或设置字符串	TextNote.Text 属性
获取或设置文本组件位置	TextNote.Coord 属性
获取或设置文本组件宽度	TextNote.Width 属性
获取或设置文本组件引线	TextNote.Leaders 属性
为文本组件添加引线	TextNote.AddLeader( )方法
删除文本组件中所有引线	TextNote.RemoveLeaders( )方法

Revit 支持两种引线: 直引线和圆弧引线, 使用 TextNoteLeaderType 枚举类型 (表 3-17) 控制 TextNote 引线。

表 3-17 引 线 类 型

功 能	成员名称	功 能	成员名称
- 添加向右圆弧引线	TNLT_ARC_R	- 添加向右引线	TNLT_STRAIGHT_R
- 添加向左圆弧引线	TNLT_ARC_L	- 添加向左引线	TNLT_STRAIGHT_L

注: 直引线和圆弧引线无法同时添加到 Text 类型。



### 3.6.5 注释符号 (Annotation Symbol)

注释符号是应用于族的符号,可唯一识别项目中的族。在 UI 中添加注释符号见图 3-45,双引线注释符号见图 3-46。



图 3-45 添加注释符号

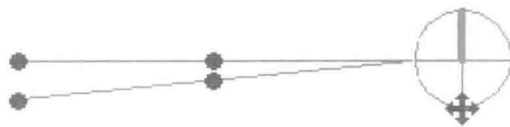


图 3-46 有两条引线的注释符号

#### 1. 创建和删除 (Create and Delete)

注释符号可以用 `Creation.Document.NewFamilyInstance()` 方法 (代码 3-48) 重载创建。

#### 代码 3-48: 新建族符号

```
public FamilyInstance NewFamilyInstance Method (XYZ origin, FamilySymbol symbol, View specView)
```

删除注释符号, 使用 `Document.Delete()` 方法。

#### 2. 添加和删除引线 (Add and Remove Leader)

添加和删除引线, 使用 `addLeader()` 和 `removeLeader()` 方法 (代码 3-49)。

#### 代码 3-49: 使用 `addLeader()` 和 `removeLeader()`

```
public void AddAndRemoveLeaders (AnnotationSymbol symbol)
{
    int leaderSize = symbol.Leaders.Size;
    string message = "Number of leaders in Annotation Symbol before add: " + leaderSize;
    symbol.addLeader();
    leaderSize = symbol.Leaders.Size;
    message += "\nNumber of leaders after add: " + leaderSize;
    symbol.removeLeader();
    leaderSize = symbol.Leaders.Size;
    message += "\nNumber of leaders after remove: " + leaderSize;

    TaskDialog.Show ("Revit", message);
}
```

## 3.7 几何 (Geometry)

Autodesk.Revit.DB 命名空间包含许多几何相关的类和图形相关的类型, 用于描述 API 中的图形表示。几何相关的类包括:

- **GeometryObject 类:** 包括由 `GeometryObject` 类派生的类。
- **Geometry Helper 类:** 包括由 `APIObject` 类派生的类和值类型。
- **Geometry Utility 类:** 包括创建非图元几何图形和查找体交集的类。



- Collection 类：包括由 IEnumerable 或 IEnumerator 接口派生的类。

在本节中，将了解如何使用各种图形相关的类型，如何从图元中检索几何数据，如何变换图元，等等。

### 3.7.1 示例：检索墙的几何数据 (Example: Retrieve Geometry Data from a Wall)

该示例阐释了如何检索墙的几何数据。包括以下信息：

- 获取墙体几何边。
- 获取墙体几何面。

注意：因为未考虑实例，该示例中从 Element 检索的几何数据是有限的。例如，示例代码中墙中包含的放样不可访问。该示例的目的就是给读者一个如何检索几何数据的基本概念，而非涵盖所有情况。有关从图元中检索几何数据的更多信息，请参阅 3.7.5 节示例：检索梁的几何数据。

#### 1. 创建几何选项 (Create Geometry Options)

为了得到墙体的几何信息，必须创建一个可提供详细精度的自定义选项 Geometry.Options 对象，见代码 3-50。

#### 代码 3-50：创建 Geometry.Options

```
Autodesk.Revit.DB.Options geomOption = application.Create.NewGeometryOptions();
if (null != geomOption)
{
    geomOption.ComputeReferences = true;
    geomOption.DetailLevel = Autodesk.Revit.DB.DetailLevels.Fine;

    // Either the DetailLevel or the View can be set, but not both
    //geomOption.View = commandData.Application.ActiveUIDocument.Document.ActiveView;

    TaskDialog.Show ("Revit", "Geometry Option created successfully.");
}
```

注：有关详细信息，请参阅 3.7.3 节几何助手类中的 Geometry.Options。

#### 2. 检索表面和边缘 (Retrieve Faces and Edges)

墙几何体是由表面和边缘组成的体。完成以下步骤以获取其表面和边缘：

- 用 Wall 类 Geometry 属性检索 Geometry.Element 实例。此实例在 Object 属性中包含所有几何对象，如体、线等。
- 迭代 Object 属性，在 Faces 和 Edges 属性中获取一个包含所有几何表面和边缘的几何体实例。
- 迭代 Faces 属性来获取所有几何表面。
- 迭代 Edges 属性来获取所有几何边缘。

示例代码见代码 3-51。

#### 代码 3-51：检索面和边

```
private void GetFacesAndEdges (Wall wall)
{
```



```
String faceInfo = "";

Autodesk.Revit.DB.Options opt = new Options( );
Autodesk.Revit.DB.GeometryElement geomElem = wall.get_Geometry( opt);
foreach ( GeometryObject geomObj in geomElem )
{
    Solid geomSolid = geomObj as Solid;
    if ( null != geomSolid )
    {
        int faces = 0;
        double totalArea = 0;
        foreach ( Face geomFace in geomSolid.Faces )
        {
            faces++;
            faceInfo += "Face " + faces + " area: " + geomFace.Area.ToString( ) + "\n";
            totalArea += geomFace.Area;
        }
        faceInfo += "Number of faces: " + faces + "\n";
        faceInfo += "Total area: " + totalArea.ToString( ) + "\n";
        foreach ( Edge geomEdge in geomSolid.Edges )
        {
            // get wall's geometry edges
        }
    }
}

TaskDialog.Show ( "Revit", faceInfo);
}
```

### 3.7.2 几何对象类 ( GeometryObject Class )

索引的属性 `Element.Geometry` 可以用来获取任何模型图元 ( 三维图元 ) 的几何形状。它既适用于系统族实例, 如墙、楼板和屋顶, 也适用于许多其他类别的族实例, 如门、窗、家具或体量。

提取的几何形状返回为 `Autodesk.Revit. DB. GeometryElement`。可以用 `GetEnumerator()` 方法遍历该图元的几何成员。

通常, 返回所提取的几何形状的顶层对象将是以下之一:

- **Solid:** 由面和边构成的边界表示。
- **Mesh:** 三角形态的三维数组。
- **Curve:** 有界的三维曲线。
- **Point:** 位于给定三维位置的可见点数据。
- **PolyLine:** 由三维点定义的一系列线段。
- **GeometryInstance:** 放置在图元内的几何图元实例。

图 3-47 示出了通过几何提取所找到的对象的层次体系。

#### 1. 曲线 ( Curves )

曲线表示 Revit 模型中的二维或三维路径。曲线可以代表整个图元的几何范围 ( 如 `CurveElements` ), 或可显示为单个图元的单条几何形状 ( 如墙或管道中心线 )。曲线和曲线





集可用作 API 中许多图元创建方法的输入。

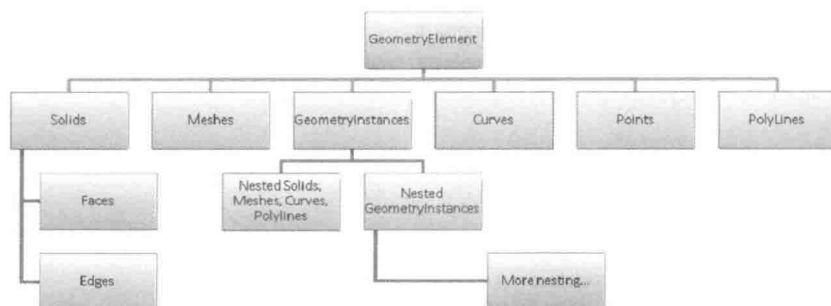


图 3-47 通过几何提取所找到的对象的层次体系

(1) 曲线分析 (Curve Analysis)。有几个 Curve 成员是适用于几何分析的工具。在某些情况下，通过快速查看它们的名称，这些 API 所能做的很可能超出用户的预期。

1) **Intersect()**。求交运算方法允许比较两个相交曲线，发现它们有何不同或相似之处。它可用所期望的方式，获得两个彼此相交曲线的一个或多个交点，还可用来识别：

- 共线直线 (段)。
- 重叠直线 (段)。
- 相同曲线。
- 无交点的完全不同曲线。

返回值识别这些不同的结果，并输出包含交叉点信息的 **IntersectionSetResult**。

2) **Project()**。投影方法将点投影到曲线上并返回有关信息：曲线上的最近点、它的参数以及到投影点的距离。

3) **Tessellate()**。将曲线分割为精确度在默认公差范围内的一系列直线段。对 **Curve.Tessellate()**，公差略大于 1/16"。此近似公差是由 Revit 充分满足显示用途所使用的内部公差。

请注意，只有直线才可能分为只有两个细分点的输出；非线性的曲线总是会输出两个以上的点，即使该曲线有一个非常大的半径，可近似等同于直线。

(2) 曲线集 (Curve Collections)。Revit API 使用不同类型的曲线集作为输入：

- **CurveLoop**：表示端点与端点连接的特定曲线系列。它可以是闭合曲环，也可以是非闭合曲环。创建方法：
  - **CurveLoop.Create()**。
  - **CurveLoop.CreateViaCopy()**。
  - **CurveLoop.CreateViaThicken()**。
- **CurveArray**：此集合类表示任意的曲线集合。使用其构造器创建。
- **CurveArrArray**：此集合类是 **CurveArrays** 的集合。使用此集合类时，该集合子元素的组织结构继承被传递的方法。例如，在 **NewExtrusion()** 中，多个 **CurveArrays** 应该代表不同的闭合曲环。

更新的 API 方法使用 .NET 的 **Curves** 集合来替代 **CurveArray** 和 **CurveArrArray**。



(3) 创建曲线 (Curve Creation)。曲线往往需要用作 Revit API 方法的输入。曲线可以由相关类的静态方法创建:

- `Line.CreateBound()`。
- `Line.CreateUnbound()`。
- `Arc.Create()`。
- `Ellipse.Create()`。
- `NurbSpline.Create()`。
- `HermiteSpline.Create()`。
- `Curve.CreateTransformed()`。

曲线创建方法应防止创建的曲线比 Revit 内部公差更短,此公差可通过查询 `Application.ShortCurveTolerance` 属性获得。

(4) 参数化曲线 (Curve Parameterization)。Revit API 中的曲线可看成一个输入参数为“ $u$ ”的数学函数,在任何给定点的三维空间曲线位置都是参数“ $u$ ”的函数。

曲线可以绑定或非绑定。非绑定曲线没有端点,代表着无限抽象或循环曲线(圆或椭圆)。

Revit 中,参数“ $u$ ”可以两种方式表示:

- “标准化”参数。起点参数值是 0.0, 终点值是 1.0。对于某些曲线类型,这使得曲线沿其长度范围的赋值很简单,例如,线的中点参数为 0.5 (注意,对于像样条线那样更复杂的曲线方程,往往不能作这样的假设)。
- “原始”参数。起点和终点参数值可以是任意值。对于给定曲线,最小和最大原始参数值可由 `Curve.GetEndParameter(int)` 获得。原始参数是有用的,因为它们的单位与 Revit 默认单位(英尺)一样。于是,为了获得一个距离曲线起点 5 英尺的位置,只要获取起点原始参数并对其加 5。原始参数也是对非绑定曲线赋值的唯一方法。

`Curve.ComputeNormalizedParameter()`和 `Curve.ComputeRawParameter()`方法在两个参数类型之间自动按比例调整大小。`Curve.IsInside()`方法可评估原始参数以查看它是否在曲线范围内。

使用参数可以求出任何给定位置曲线的各种属性值:

- 给定曲线的空间位置。这由 `Curve.Evaluate()` 返回。可提供原始参数或标准化参数。如果还调用了 `ComputeDerivatives()`,也是由该方法返回的 `Transform` 的 `.Origin` 属性。
- 给定曲线的一阶导数/切线向量。这是由 `Curve.ComputeDerivatives()` 返回的 `Transform` 的 `.BasisX` 属性。
- 给定曲线的二阶导数/法线向量。这是由 `Curve.ComputeDerivatives()` 返回的 `Transform` 的 `.BasisY` 属性。
- 给定曲线的副法线向量,定义为切线与法线向量的叉积。这是由 `Curve.ComputeDerivatives()` 返回的 `Transform` 的 `.BasisZ` 属性。

返回的所有向量都是非标准化的(但可以在 Revit API 中用 `XYZ.Normalize()` 标准化任何向量)。请注意,当曲线为直线时,不会有法线和副法线向量值设置,但可以用切线向量计算出给定平面内直线的法线向量。



API 示例 “DirectionCalculation” 使用墙位置曲线的切线向量找出面朝南的外墙, 运行结果如图 3-48。

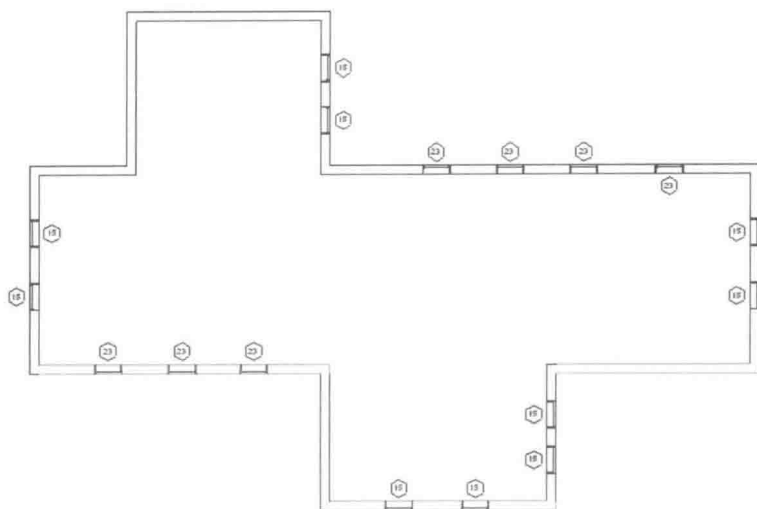


图 3-48 找出并高亮显示面朝南的外墙

(5) 曲线类型 (Curve Types)。Revit 使用多种类型的曲线来表示文件中的曲线几何形状, 见表 3-18。

表 3-18

Revit 中的曲线类型

曲线类型	Revit API 类	定义	注 释
线段	Line	由端点定义的线段	由 Curve.GetEndpoint() 获取端点
直线	Line	由位置和方向定义的无穷线	用 Curve.IsBound 鉴别。 通过原始参数=0 处的点坐标及切线向量来找出线方程的输入参数
弧	Arc	绑定的圆弧	以一定角度起始、结束。这些角度可由圆弧每个端点的原始参数值获取
圆	Arc	非绑定的圆	用 Curve.IsBound 识别。 用原始参数计算 ( $0 \sim 2\pi$ )
柱面螺旋线	CylindricalHelix	与圆柱轴成常量角度绕圆柱面的螺旋线	仅用于楼梯、栏杆扶手等应用, 访问其他 Revit 图元和几何曲线时不会使用或遇到
椭圆弧	Ellipse	绑定的椭圆线段	
椭圆	Ellipse	非绑定的椭圆	用 Curve.IsBound 识别。用原始参数计算 ( $0 \sim 2\pi$ )
非均匀有理 B 样条	NurbSpline	非均匀有理 B 样条	用于各种 Revit 工具草绘的样条线, 外加输入的几何形状
厄米样条	HermiteSpline	点集之间插值的样条曲线	用作工具, 如由点生成曲线和柔性风管/管道, 外加输入的几何形状

所有 Revit 曲线类型的数学表达式都可在下述内容中找到。

(6) 曲线类型的数学表达式 (Mathematical Representations of Curve Types)。本节描述



Revit 几何图形中遇到的曲线类型、它们的属性，以及它们的数学表达式。

1) 线段 (Bound Lines)。线段由其端点定义。在 Revit API 中，从 Curve 层级的 GetEndPoint() 方法获取线的端点。

依据标准化参数 “ $u$ ” 和线端点，线段上点的方程为

$$P(u) = P_1 + u(P_2 - P_1)$$

2) 直线 (Unbound Lines)。Revit API 对直线作专门处理。大多数曲线属性不能使用，但是，在提供了原始参数的情况下，Evaluate() 和 ComputeDerivatives() 可用来获取沿曲线方向的位置。

依据原始参数 “ $u$ ” 和线的原点及标准化的方向向量，直线上点的方程为

$$P(u) = P_0 + uV$$

3) 圆弧和圆 (Arcs and Circles)。Revit API 中圆弧和圆由 Arc 类表示。它们由圆弧半径、中心和圆弧平面的法线向量确定，在 Revit API 中可作为属性直接从 Arc 类访问。

圆的 IsBound 属性设置为 true。这意味着它们只能用原始参数 (范围为  $0 \sim 2\pi$ ) 进行计算，依据原始参数的圆上点的方程为

$$P(u) = C + r(n_x \cos u + n_y \sin u)$$

这里假定圆位于  $XY$  平面内。

弧以某个角度开始和结束。这些角度可由弧的每个端点的原始参数值获取，介于其间的角度值可以插入到上述同一方程中。

4) 柱面螺旋线 (Cylindrical Helixes)。Revit API 中，柱面螺旋线是由 CylindricalHelix 类表示。依据以下参数来定义：螺旋线所围绕的圆柱轴的基点、半径、 $x$  和  $y$  向量、间距、起始角和终止角。

5) 椭圆和椭圆弧 (Ellipse and Elliptical Arcs)。Revit API 中，椭圆和椭圆弧段由 Ellipse 类表示。类似于圆和圆弧，它们依据其给定平面的  $X$  轴和  $Y$  轴半径、中心及椭圆平面法线向量来定义。

完整椭圆 IsBound 属性设置为 true。类似于圆，它们可以通过  $0 \sim 2\pi$  之间的原始参数进行计算：

$$P(u) = C + n_x r_x \cos u + n_y r_y \sin u$$

6) 非均匀有理 B 样条 (NurbSpline)。NURBS (非均匀有理 B 样条) 用于用户草绘的样条线，作为曲线或三维对象草图的某些部分。它们也可用来表示某些类型的几何数据输入。

非均匀有理 B 样条的数据包括：

- 控制点数组，数组长度为  $n+1$ 。
- 权重数组，长度亦为  $n+1$ 。
- 曲线次数，其值为曲线阶数 ( $k$ ) 减一。
- 节点向量，长度为  $n+k+1$ 。

$$P(u) = \frac{\sum_{i=0}^n P_i w_i N_{i,k}(u)}{\sum_{i=0}^n w_i N_{i,k}(u)}, \quad 0 \leq u \leq u_{\max}$$



NurbSplines 作为 Revit 的草绘工具, 可由独立算法得到的控制点和阶次生成。Revit 算法所执行的计算可被外部复制, 请参阅下面示例:

```
NurbSplinespline = curve.GeometryCurve as NurbSpline;
```

```
DoubleArrayknots = spline.Knots;
```

```
// Convert to generic collection
```

```
List<double> knotList = new List<double>();
```

```
for (int i = 0; i < knots.Size; i++)
```

```
{
```

```
    knotList.Add (knots.get_Item (i));
```

```
}
```

```
// Preparation - get distance between each control point
```

```
IList<XYZ> controlPoints = spline.CtrlPoints;
```

```
int numControlPoints = controlPoints.Count;
```

```
double [] chordLengths = new double [numControlPoints - 1];
```

```
for (int iControlPoint = 1; iControlPoint < numControlPoints; ++iControlPoint)
```

```
{
```

```
    double chordLength =
```

```
        controlPoints [iControlPoint] .DistanceTo (controlPoints [iControlPoint - 1]);
```

```
    chordLengths [iControlPoint - 1] = chordLength;
```

```
}
```

```
int degree = spline.Degree;
```

```
int order = degree + 1;
```

```
int numKnots = numControlPoints + order;
```

```
double [] computedKnots = new double [numKnots];
```

```
int iKnot = 0;
```

```
// Start knot with multiplicity degree + 1.
```

```
double startKnot = 0.0;
```

```
double knot = startKnot;
```

```
for (iKnot = 0; iKnot < order; ++iKnot)
```

```
{
```

```
    computedKnots [iKnot] = knot;
```

```
}
```

```
// Interior knots based on chord lengths
```



```

double prevKnot = knot;

for (/*blank*/; iKnot <= numControlPoints; ++iKnot)
    // Last loop computes end knot but does not set interior knot.
    {
        double knotIncrement = 0.0;
        for (int jj = iKnot - order; jj < iKnot - 1; ++jj)
        {
            knotIncrement += chordLengths [jj];
        }

        knotIncrement /= degree;
        knot = prevKnot + knotIncrement;
        if (iKnot < numControlPoints)
            computedKnots [iKnot] = knot;
        else
            break;    // Leave "knot" set to the end knot; do not increment "ii".

        prevKnot = knot;
    }

// End knot with multiplicity degree + 1.
for (/*blank*/; iKnot < numKnots; ++iKnot)
{
    computedKnots [iKnot] = knot;
}

```

7) 厄米样条 (HermiteSpline)。厄米样条用于在一组控制点之间插值的曲线，如由点生成的曲线和 MEP 中柔性风管及管道。它们也可用来表示某些类型的几何数据输入。Revit API 中，HermiteSpline 类可通过 ControlPoints、Tangents 和 Parameters 属性访问点数组、切线向量及参数。

两个节点之间的 Hermite 样条曲线方程为

$$P(u) = h_{00}(u)P_k + (u_{k+1} - u_k)h_{10}(u)M_k + h_{01}(u)P_{k+1} + (u_{k+1} - u_k)h_{11}(u)M_{k+1}$$

$P_k$ 、 $P_{k+1}$  表示各个节点， $M_k$ 、 $M_{k+1}$  表示切线向量， $u_k$ 、 $u_{k+1}$  表示节点参数，基本函数为：

$$h_{00}(u) = 2u^3 - 3u^2 + 1, h_{10}(u) = u^3 - 2u^2 + u$$

$$h_{01}(u) = -2u^3 + 3u^2$$

$$h_{11}(u) = u^3 - u^2$$



## 2. 几何实例 (GeometryInstances)

**GeometryInstance** 表示 Revit 以默认配置存储的一组几何图形, 作为图元属性的结果转换到合适的位置。最常见的情况是在族实例中遇到几何实例。Revit 用 **GeometryInstance** 作为给定族存储几何图形的单个副本, 并在多个实例中重复使用。

要注意的是, 并非所有 **Family** 实例都包含 **GeometryInstance**。当 Revit 需要为给定实例生成族几何图形的唯一副本 (由于局部连接、相交和实例放置相关的其他因素影响) 时, 将不包含 **GeometryInstance**; 代之以 **Solid** 出现在层次结构的顶层。

通过使用 **GetSymbolGeometry()** 和 **GetInstanceGeometry()** 方法, **GeometryInstance** 可以读取它的几何图形。正如顶层返回一样, 这些方法返回可解析的另一个 **Autodesk. Revit.DB. GeometryElement**。

**GetSymbolGeometry()** 返回以族坐标系统表示的几何图形。例如, 当要获得一张“通用型”桌子图片, 而不关心其在项目中的方向和定位时, 可使用该方法。这也是返回 Revit 实际几何对象而非副本的唯一重载方法。这很重要, 因为用此几何形状作为输入的操作来创建其他图元 (如尺寸标注或放置基于面的族) 需要参照原始几何图形。

**GetInstanceGeometry()** 返回以实例所在项目坐标系统表示的几何形状。例如, 当要获得一幅项目中某个特定实例的几何图形时, 可使用该方法 (如要确保那些桌子平行于房间的墙壁)。该方法总是返回图元几何图形的副本, 因此它适合作为输出成果或几何分析工具的实现, 而不适合以此几何图形作为参照来创建其他 Revit 图元。

**GetInstanceGeometry()** 和 **GetSymbolGeometry()** 都还有其重载方法, 可通过任意坐标系统对几何图形作变换, 见图 3-49。这些方法总是返回副本, 类似于 **GetInstanceGeometry()**。

**GeometryInstance** 还存储符号坐标空间到实例坐标系的变换。此变换可作为 **Transform** 属性进行访问。当通过 **GetInstanceGeometry()** 提取几何形状副本时, 也使用此变换。有关详细信息, 请参阅 3.7.3 节中的变换 (**Transform**)。

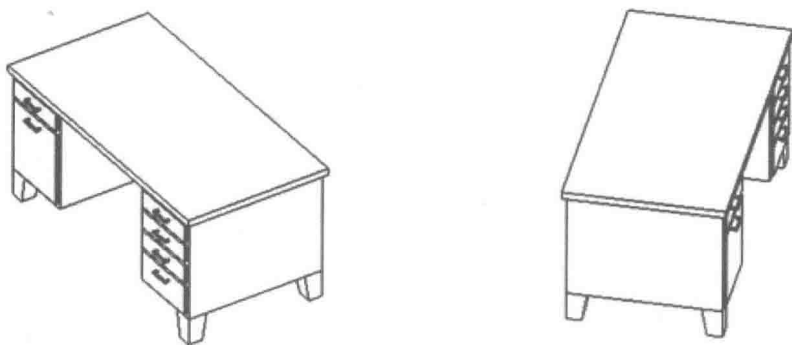


图 3-49 用不同变换放置的两个族实例 (从两者都可获得相同的几何图形)

对某些族, 实例可以嵌套多个层级。倘若遇到嵌套实例, 则以类似于作为顶层实例的方式来解析它。

提供以下两个示例来说明如何解析实例的几何图形。

在代码 3-52 中, 从几何实例的 **GetInstanceGeometry()** 方法提取曲线。



代码 3-52: 从实例中获取曲线

```

public void GetAndTransformCurve (Autodesk.Revit.ApplicationServices.Application app,
    Autodesk.Revit.DB.Element element, Options geoOptions)
{
    // Get geometry element of the selected element
    Autodesk.Revit.DB.GeometryElement geoElement = element.get_Geometry (geoOptions);

    // Get geometry object
    foreach (GeometryObject geoObject in geoElement)
    {
        // Get the geometry instance which contains the geometry information
        Autodesk.Revit.DB.GeometryInstance instance =
            geoObject as Autodesk.Revit.DB.GeometryInstance;
        if (null != instance)
        {
            GeometryElement instanceGeometryElement = instance.GetInstanceGeometry();
            foreach (GeometryObject o in instanceGeometryElement)
            {
                // Try to find curves
                Curve curve = o as Curve;
                if (curve != null)
                {
                    // The curve is already transformed into the project coordinate system
                }
            }
        }
    }
}

```

在代码 3-53 中, 使用 `GetSymbolGeometry()` 从实例中获取体信息。然后用 `GeometryInstance.Transform` 将其构成点变换到项目坐标系。

代码 3-53: 从实例中获取体信息

```

private void GetAndTransformSolidInfo (Application application, Element element, Options geoOptions)
{
    // Get geometry element of the selected element
    Autodesk.Revit.DB.GeometryElement geoElement = element.get_Geometry (geoOptions);

    // Get geometry object
    foreach (GeometryObject geoObject in geoElement)
    {
        // Get the geometry instance which contains the geometry information
        Autodesk.Revit.DB.GeometryInstance instance =
            geoObject as Autodesk.Revit.DB.GeometryInstance;
        if (null != instance)
        {
            GeometryElement instanceGeometryElement = instance.GetSymbolGeometry();
            foreach (GeometryObject instObj in instanceGeometryElement)
            {

```





```
Solid solid = instObj as Solid;
if (null == solid || 0 == solid.Faces.Size || 0 == solid.Edges.Size)
{
    continue;
}

Transform instTransform = instance.Transform;
// Get the faces and edges from solid, and transform the formed points
foreach (Face face in solid.Faces)
{
    Mesh mesh = face.Triangulate();
    foreach (XYZ ii in mesh.Vertices)
    {
        XYZ point = ii;
        XYZ transformedPoint = instTransform.OfPoint (point);
    }
}
foreach (Edge edge in solid.Edges)
{
    foreach (XYZ ii in edge.Tessellate())
    {
        XYZ point = ii;
        XYZ transformedPoint = instTransform.OfPoint (point);
    }
}
}
```

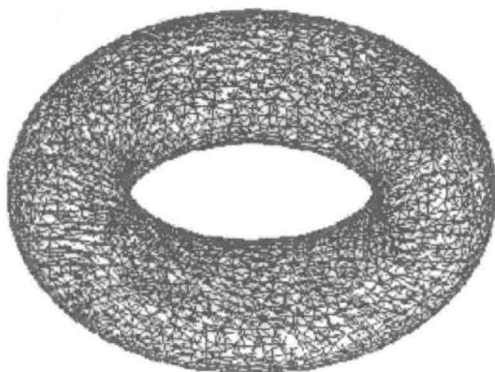


图 3-50 表示圆环体的网格

注：有关检索族实例几何图形的详细信息，请参阅 3.7.5 节示例：检索梁的几何数据。

### 3. 网格 (Meshes)

网格是共同形成三维形状的三角形边界的集合，图 3-50 是圆环体的网格。通常，若图元是由某些导入操作所创建的，则会在这些 Revit 图元几何图形中存在网格；某些本地 Revit 图元也会用到网格，如 TopographySurface (地形表面)。还可以对任何给定的 Revit 面调用 Face.Triangulate() 获取网格。

代码 3-54 阐释了如何得到以网格表示 Revit 面的几何形状。

#### 代码 3-54：提取网格几何图形

```
private void GetTrianglesFromFace (Face face)
{
    // Get mesh
```



```

Mesh mesh = face.Triangulate();
for (int i = 0; i < mesh.NumTriangles; i++)
{
    MeshTriangle triangle = mesh.get_Triangle(i);
    XYZ vertex1 = triangle.get_VerTEX(0);
    XYZ vertex2 = triangle.get_VerTEX(1);
    XYZ vertex3 = triangle.get_VerTEX(2);
}
}

```

注：当构建网格时，Revit 用于显示的容许公差通过重载无参数 `Triangulate()` 方法获得。`Triangulate()` 重载还可采用一个 0~1 之间的双精度数来设置详细程度。0 表示“粗略”，1 表示“精细”。

#### 4. 点 (Points)

点表示三维空间中的一个可见坐标。这通常会在体量族图元中遇到，如 `ReferencePoint`。`Point` 类提供对点坐标的读访问，以及获取用作其他函数输入的点参照。

#### 5. 多段线 (PolyLines)

多段线是由一组坐标点定义的线段集合。在导入的几何图形中通常会遇到这些多段线。`PolyLine` 类提供了读取这些坐标点的能力：

- `PolyLine.NumberOfCoordinates()`：多段线上点的数量。
- `PolyLine.GetCoordinate()`：由索引获取坐标。
- `PolyLine.GetCoordinates()`：获取多段线所有坐标点的集合。
- `PolyLine.Evaluate()`：给定一个标准化的参数（从 0 到 1）沿整个多段线长度范围计算出一个三维空间点。

#### 6. 体、表面和边缘 (Solids、Faces 和 Edges)

`Solid` 是一个表示表面和边缘的集合的 Revit API 对象。在 Revit 中，这些集合通常是完全封闭的体量，但也会遇到壳体或局部有界体量。要注意的是，有时 Revit 几何图形会包含“0”边缘和表面的未使用体，检查 `Edges` 和 `Faces` 成员可滤掉这些体。

Revit API 可以读取面和边的集合，并可以计算体的表面积、体积和质心。

(1) 参数化边缘和表面 (Edge and Face Parameterization)。边缘是给定表面的边界曲线。

使用 `EdgeLoops` 属性迭代 `Face` 的所有边缘。每个循环都表示表面的一个封闭边界。边缘总是参数化为 0~1 的数值。使用 `Edge.AsCurve()` 和 `Edge.AsCurveFollowingFace()` 函数，可以获取 `Edge` 的 `Curve` 表示。

通常，边缘由求出两个表面的相交线来定义。但 Revit 在绘图时并不重新计算此交线，因此，边缘存储为点的列表——直边为端点列表而曲边为细分点列表。这些点可由 `Tessellated()` 方法获得。

剖面会产生“剖切边缘”，这些都是“人为的”边，而不是模型级几何图形的一部分，因此不提供 `Reference`。

(2) 边缘方向 (Edge Direction)。“第一个”表面的方向通常是顺时针方向（“第一个”表示 Revit 已为特定边缘识别的任意表面）。但由于两个不同的表面交于某个特定边缘，不管哪个面被关注，边缘都具有相同的参数方向，有时需要计算出特定表面上边缘的方向。



图 3-51 说明了它是如何使用的。对于 Face 0, 所有的边都是参数化的顺时针方向。对于 Face 1, 与 Face 0 的共边未重新参数化, 因此就 Face 1 而言, 其边缘是相反的方向, 某些边缘相交处两个边缘的参数都为 0 (或 1)。

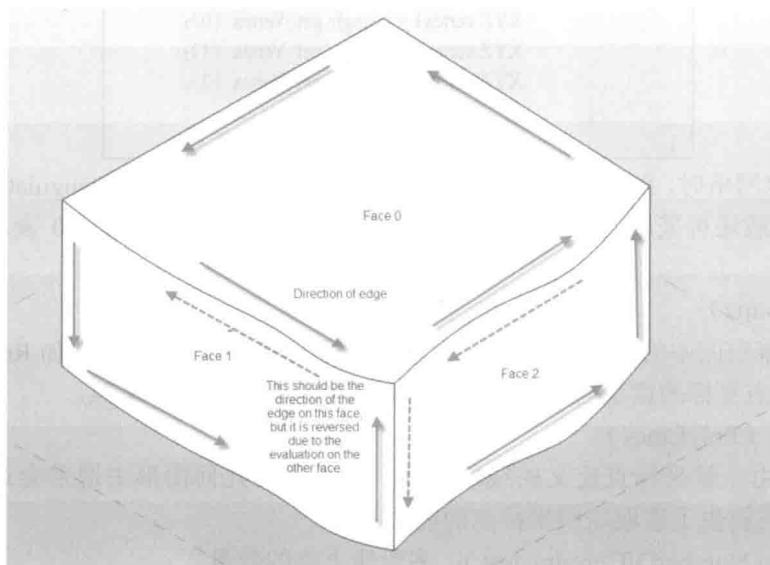


图 3-51 参数化边缘

SDK 中示例 “PanelEdgeLengthAngle” (结果见图 3-52) 说明了如何识别给定表面的反向边缘。使用边缘端点切线向量来计算相邻边缘之间的夹角, 并检测是否要翻转每个相交的切线向量以计算出正确的角度。

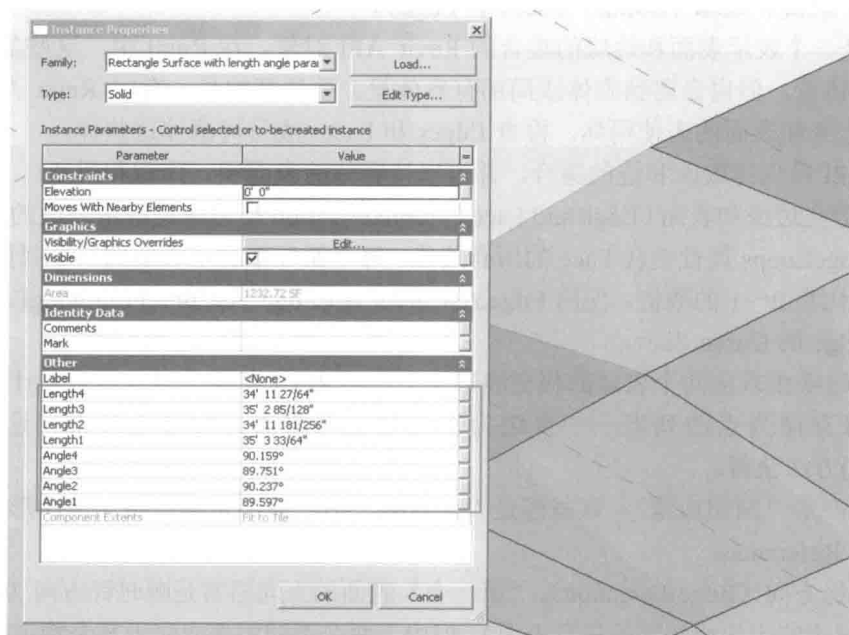


图 3-52 PanelEdgeLengthAngle 结果



(3) 表面 (Faces)。Revit API 中 Faces 可以表示为两个输入参数“ $u$ ”和“ $v$ ”的数学函数，在三维空间中任意给定点的表面位置为这两个参数的函数。 $U$ 、 $V$  方向基于给定表面的形状自动确定。 $U$ 、 $V$  方向可表示为表面上的网格线，见图 3-53。

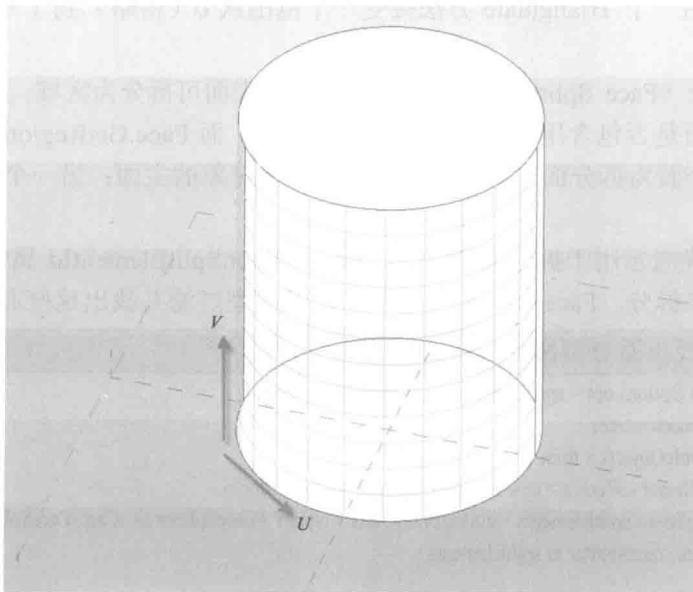


图 3-53 圆柱面上的  $UV$  网格线

使用  $UV$  参数可以计算表面在任何给定位置的各种属性：

- 确定参数是否在表面的边界内，使用 `Face.IsInside()`。
- 指定  $UV$  参数值所给定表面的空间位置。它由 `Face.Evaluate()` 返回。如果还调用了 `ComputeDerivatives()`，该方法还会返回 Transform 的 `.Origin` 属性。
- 给定表面在  $U$  方向上的切线向量。这是由 `Face.ComputeDerivatives()` 返回的 Transform 的 `.BasisX` 属性。
- 给定表面在  $V$  方向上的切线向量。这是由 `Face.ComputeDerivatives()` 返回的 Transform 的 `.BasisY` 属性。
- 给定表面的法线向量。这是由 `Face.ComputeDerivatives()` 返回的 Transform 的 `.BasisZ` 属性。

所有返回的向量都是非标准化的。

(4) 表面分析 (Face Analysis)。Face 包含的一些方法为适用于几何分析的工具。

1) `Intersect()`。`Intersect` 方法计算表面和曲线之间的交集。它可用于确定：

- 两个对象之间的交点。
- 距离交点最近的边缘（假设该位置附近有一条边）。
- 曲线与表面完全重合。
- 曲线和表面不相交。

2) `Project()`。`Project` 方法将一个点投影到输入的面上，并返回投影点的信息、到面的



距离、距离投影点最近的边。

3) **Triangulate()**。**Triangulate** 方法获取表面的三角形近似网格。此方法有两个重载。无参数方法类似于 **Curve.Tessellate()**，其网格点的精度在 Revit 所使用的输入公差（略大于 1/16"）范围内。另一个 **Triangulate** 方法接受一个范围从 0（粗略）到 1（精细）作为详细程度的参数。

(5) 表面拆分 (**Face Splitting**)。用拆分面命令，表面可拆分为区域。**Face.HasRegions** 属性会报告某个面是否包含用拆分面命令创建的区域，而 **Face.GetRegions()** 方法将返回一个面的列表，一个面为拆分面（如楼层的墙体）主体对象的主面；另一个面为每个拆分面区域。

**FaceSplitter** 类表示用于拆分面的图元。**FaceSplitter.SplitElementId** 属性提供图元 ID，其表面由此图元来拆分。**FaceSplitter** 类可用于通过类型过滤并找出这些面，见代码 3-55。

#### 代码 3-55：找出面分割图元

```
Autodesk.Revit.DB.Options opt = app.Create.NewGeometryOptions();
opt.ComputeReferences = true;
opt.IncludeNonVisibleObjects = true;
FilteredElementCollector collector = new FilteredElementCollector (doc);
ICollection<FaceSplitter> splitElements = collector.OfClass (typeof (FaceSplitter)) .Cast<FaceSplitter>().ToList();
foreach (FaceSplitter faceSplitter in splitElements)
{
    Element splitElement = doc.GetElement (faceSplitter.SplitElementId);
    Autodesk.Revit.DB.GeometryElement geomElem = faceSplitter.get_Geometry (opt);
    foreach (GeometryObject geomObj in geomElem)
    {
        Line line = geomObj as Line;
        if (line != null)
        {
            XYZ end1 = line.GetEndPoint (0);
            XYZ end2 = line.GetEndPoint (1);
            double length = line.ApproximateLength;
        }
    }
}
```

(6) 表面类型 (**Face Types**)。Revit 使用多种曲面类型来表示文件中的面几何形状，包括表 3-19 中的曲面类型。

表 3-19

Revit 中的曲面类型

面类型	Revit API 类	定义	注 释
平面	PlanarFace	平面由原点和 <i>UV</i> 单位向量确定	
圆柱面	CylindricalFace	沿轴拉伸圆所定义的面	.Radius 提供 “半径向量”——圆的单位向量乘以半径值
圆锥面	ConicalFace	绕轴旋转一条线所确定的面	.Radius 提供 “半径向量”——圆的单位向量乘以半径值



续表

面类型	Revit API 类	定义	注 释
旋转面	RevolvedFace	绕轴旋转一条任意曲线所确定的面	.Radius 提供旋转面的单位向量, 与“半径”无关
直纹曲面	RuledFace	在两条轮廓线之间或一条轮廓曲线和一个点之间, 放样一根由直线所形成的面	曲线和点均可作为属性获取
厄米面	HermiteFace	由点间厄米插值所形成的面	

所有 Revit 面类型的数学表达式都可以在下述内容中找到。

(7) 面类型的数学表达式 (Mathematical Representation of Face Types)。本节描述 Revit 几何图形中遇到的面类型、它们的属性, 以及它们的数学表达式。

1) 平面 (PlanarFace)。由原点和  $UV$  单位向量确定的平面, 其参数方程为

$$P(u, v) = P_0 + un_u + vn_v$$

2) 圆柱面 (CylindricalFace)。沿轴拉伸圆所定义的面, Revit API 提供了以下属性:

- 面的原点。
- 拉伸轴。
- $X$  和  $Y$  “半径向量”。这些向量等于圆的单位向量乘以圆的半径。要注意的是, 单位向量可能表现为右手或左手控制框架。

圆柱面的参数方程为

$$P(u, v) = P_0 + r_x \cos(u) + r_y \sin(u) + vn_{axis}$$

3) 圆锥面 (ConicalFace)。绕轴旋转一条线所定义的面。Revit API 提供了以下属性:

- 面的原点。
- 圆锥轴。
- $X$  和  $Y$  “半径向量”。这些向量等于单位向量乘以旋转所形成的圆的半径。请注意, 单位向量可能表现为右手或左手控制框架。
- 面的半角。

圆锥面的参数方程为

$$P(u, v) = P_0 + v[\sin(a)(r_x \cos(u) + r_y \sin(u)) + \cos(a)n_{axis}]$$

4) 旋转面 (RevolvedFace)。任意曲线绕轴旋转所定义的面。Revit API 提供了以下属性:

- 面的原点。
- 面的轴。
- 轮廓曲线。
- 旋转曲线的单位向量 (称为“半径”是错误的)。

旋转面的参数方程为

$$P(u, v) = P_0 + C(v)[r_x \cos(u) + r_y \sin(u) + n_{axis}]$$

5) 直纹曲面 (RuledFace)。直纹曲面通过放样两个轮廓曲线之间或曲线与点之间的直线来创建。Revit API 提供了曲线与点属性。



直纹面的参数方程为

$$P(u, v) = C_1(u) + v[C_2(u) - C_1(u)]$$

上述方程为两条有效曲线的情况。如果其中一条曲线以点代替, 则方程简化为:

$$P(u, v) = P_1 + v[C_2(u) - P_1] \quad \text{或} \quad P(u, v) = C_1(u) + v[P_2 - C_1(u)]$$

如果没有曲线及两个点的直纹曲面是退化的面, 将不予返回。

6) 厄米面 (HermiteFace)。三次 Hermite 样条面。Revit API 提供了:

- 用作样条插值点的  $u$ 、 $v$  参数数组。
- 一个含有每个节点上三维点的数组 (组织形式为先增加  $u$ , 后增加  $v$ )。
- 一个含有每个节点切线向量的数组。
- 一个含有每个节点扭向量的数组。

节点  $(u_1, v_1)$  和  $(u_2, v_2)$  之间的厄米面的参数表达式为

$$P(u, v) = U^T [M_H] [B] [M_H]^T V$$

$U = [u^3 \ u^2 \ u_1]^T$ ,  $V = [v^3 \ v^2 \ v_1]^T$ ,  $M_H$  为厄米矩阵:

$$[M_H] = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$B$  是由插值点的面属性得到的系数矩阵:

$$[B] = \begin{bmatrix} P_{u_1 v_1} & P_{u_1 v_2} & P'_{v(u_1 v_1)} & P'_{v(u_1 v_2)} \\ P_{u_2 v_1} & P_{u_2 v_2} & P'_{v(u_2 v_1)} & P'_{v(u_2 v_2)} \\ P'_{u(u_1 v_1)} & P'_{u(u_1 v_2)} & P'_{uv(u_1 v_1)} & P'_{uv(u_1 v_2)} \\ P'_{u(u_2 v_1)} & P'_{u(u_2 v_2)} & P'_{uv(u_2 v_1)} & P'_{uv(u_2 v_2)} \end{bmatrix}$$

(8) 体分析 (Solid Analysis)。方法 Solid.IntersectWithCurve() 计算闭合体量体和曲线之间的交叉。SolidCurveIntersectionOptions 类可指定在 IntersectWithCurve() 方法得到的结果中包含的曲线段是在体量内部还是外部。体内部曲线段包括与体表面重合的曲线段。曲线段及其参数均可在结果中获得。

代码 3-56 使用 IntersectWithCurve() 方法来计算柱子中的钢筋长度。

#### 代码 3-56: 找出体与曲线的交叉

```
void FindColumnRebarIntersections (Document document, FamilyInstance column)
{
    // Find rebar hosted by this column
    RebarHostData rebarHostData = RebarHostData.GetRebarHostData (column);
    if (null != rebarHostData)
    {
        IList<Rebar> rebars = rebarHostData.GetRebarsInHost();
        if (rebars.Count > 0)
        {
            Options geomOptions = new Options();
            geomOptions.ComputeReferences = true;
```



```

geomOptions.DetailLevel = ViewDetailLevel.Fine;
GeometryElement geomElement = column.get_Geometry (geomOptions);
foreach (GeometryObject geomObj in geomElement)
{
    GeometryInstance geomInst = geomObj as GeometryInstance;
    if (null != geomInst)
    {
        GeometryElement columnGeometry = geomInst.GetInstanceGeometry();
        foreach (GeometryObject obj in columnGeometry)
        {
            Solid solid = obj as Solid;
            if (null != solid)
            {
                SolidCurveIntersectionOptions options = new SolidCurveIntersectionOptions();
                foreach (Rebar rebar in rebars)
                {
                    // Get the centerlines for the rebar to find their intersection with the column
                    IList<Curve> curves = rebar.GetCenterlineCurves (false, false, false);
                    foreach (Curve curve in curves)
                    {
                        SolidCurveIntersection intersection = solid.IntersectWithCurve (curve, options);
                        for (int n = 0; n < intersection.SegmentCount; n++)
                        {
                            // calculate length of rebar that is inside the column
                            Curve curveInside = intersection.GetCurveSegment (n);
                            double rebarLengthinColumn = curveInside.Length;
                        }
                    }
                }
            }
        }
    }
}

```

(9) 创建体和面 (Solid and Face Creation)。体和面有时会用作其他实用程序的输入。Revit API 提供了几个程序例子，可用于从头开始创建或者从其他输入得到这种几何形状。

#### 1) 变换几何图形 (Transformed Geometry)。

- **GeometryElement.GetTransformed()**。应用变换返回所输入几何图元的副本。由于此几何图形是个副本，因此其成员不能用作其他 Revit 图元的输入参照，但可以用于几何分析与提取。

2) 几何创建实用程序 (Geometry Creation Utilities)。GeometryCreationUtilities 类是个实用程序类，可以构建基本的实体形状：

- **Extrusion (拉伸)**。





- Revolution (旋转)。
- Sweep (放样)。
- Blend (融合)。
- SweptBlend (放样融合)。

此类生成的几何图形并未作为任何图元的一部分添加到文件中。不过, 所创建的体可以用作其他 API 函数的输入, 包括:

- 在分析可视化框架内作为方法的输入面 (SpatialFieldManager.AddSpatialFieldPrimitive()) —— 允许用户将所创建的相关文件中其他图元的形状可视化。
- 作为输入体通过相交找出三维图元。
- 作为布尔操作的一个或多个输入。
- 作为几何计算的一部分 (例如, 使用 Face.Project()、Face.Intersect(), 或其他的 Face、Solid 和 Edge 的几何方法)。

代码 3-57 使用 GeometryCreationUtilities 类来创建基于位置和高度圆柱状形状。例如, 可用于围绕墙端部创建一个体量, 以便找到离墙端点很近的其他墙。

代码 3-57: 创建圆柱状体

```
private Solid CreateCylindricalVolume (XYZ point, double height, double radius)
{
    // build cylindrical shape around endpoint
    List<CurveLoop> curveloops = new List<CurveLoop>();
    CurveLoop circle = new CurveLoop();

    // For solid geometry creation, two curves are necessary, even for closed
    // cyclic shapes like circles
    circle.Append (m_app.Create.NewArc (point, radius, 0, Math.PI, XYZ.BasisX, XYZ.BasisY));
    circle.Append (m_app.Create.NewArc (point, radius, Math.PI, 2 * Math.PI, XYZ.BasisX, XYZ.BasisY));
    curveloops.Add (circle);

    Solid createdCylinder = GeometryCreationUtilities.CreateExtrusionGeometry (curveloops, XYZ.BasisZ, height);

    return createdCylinder;
}
```

3) 布尔操作 (Boolean Operations)。BooleanOperationsUtils 类提供了合并一对体几何对象的方法。

ExecuteBooleanOperation()方法采用体的副本作为输入, 生成一个新体作为结果。其第一个参数可以是任何体, 既可从 Revit 图元直接获得, 也可以通过别的操作, 如 GeometryCreationUtils 来创建。

ExecuteBooleanOperationModifyingOriginalSolid()方法直接在第一个输入的体上执行布尔运算。第一个输入必须为一个不是直接从 Revit 图元获得的体。属性 GeometryObject.IsElementGeometry 可以识别体是否适合作为此方法的输入。

这两种方法的选项包括这些操作类型: Union (并集)、Difference (差集)、Intersect



(交集)。代码 3-58 演示了如何获取两个体的交集，并找出其体积。

**代码 3-58: 体交集的体积**

```
private void ComputeIntersectionVolume (Solid solidA, Solid solidB)
{
    Solid intersection = BooleanOperationsUtils.ExecuteBooleanOperation (solidA, solidB, BooleanOperationsType.Intersect);
    double volumeOfIntersection = intersection.Volume;
}
```

### 3.7.3 几何助手类 (Geometry Helper Classes)

API 中有一些 Geometry Helper 类。Helper 类用于描述某些图元的几何信息，例如，使用 BoundingBoxXYZ 类为视图定义一个 CropBox (裁剪框)。

- BoundingBoxXYZ: 三维矩形框，用于诸如定义一个三维视图剖面区域等情况。
- Transform: 变换三维仿射空间。
- Reference: 对 Revit 模型几何对象的一个固定的参照，用于创建诸如尺寸之类的图元。
- Plane: 几何体中的平表面。
- Options: 用于解析几何体的用户选项。
- XYZ: 表示三维空间坐标的对象。
- UV: 表示二维空间坐标的对象。
- BoundingBoxUV: 平行于坐标轴的二维矩形。

#### 1. 变换 (Transform)

Revit 应用程序中，变换限定为  $3 \times 4$  的变换 (Matrix)，在模型空间中变换一个对象相对于其他模型空间和其他对象的位置。变换由模型空间中的位置和方向来建立。三个方向向量 BasisX、BasisY 和 BasisZ 属性以及 Origin 原点提供了变换的所有信息。由四个值组成的矩阵如下：

$$\begin{pmatrix} \text{BasisX.X} & \text{BasisY.X} & \text{BasisZ.X} & \text{Origin.X} \\ \text{BasisX.Y} & \text{BasisY.Y} & \text{BasisZ.Y} & \text{Origin.Y} \\ \text{BasisX.Z} & \text{BasisY.Z} & \text{BasisZ.Z} & \text{Origin.Z} \end{pmatrix}$$

对点应用变换如下：

$$\begin{pmatrix} \text{BasisX.X} & \text{BasisY.X} & \text{BasisZ.X} & \text{Origin.X} \\ \text{BasisX.Y} & \text{BasisY.Y} & \text{BasisZ.Y} & \text{Origin.Y} \\ \text{BasisX.Z} & \text{BasisY.Z} & \text{BasisZ.Z} & \text{Origin.Z} \end{pmatrix} \times \begin{pmatrix} \text{XYZ.X} \\ \text{XYZ.Y} \\ \text{XYZ.Z} \\ 1 \end{pmatrix}$$

Transform OfPoint 方法实现上述功能，代码 3-59 是一个相同变换过程的示例。

**代码 3-59: 变换示例**

```
public static XYZ TransformPoint (XYZ point, Transform transform)
{
    double x = point.X;
```



```

double y = point.Y;
double z = point.Z;

//transform basis of the old coordinate system in the new coordinate // system
XYZ b0 = transform.get_Basis (0);
XYZ b1 = transform.get_Basis (1);
XYZ b2 = transform.get_Basis (2);
XYZ origin = transform.Origin;

//transform the origin of the old coordinate system in the new
//coordinate system
double xTemp = x * b0.X + y * b1.X + z * b2.X + origin.X;
double yTemp = x * b0.Y + y * b1.Y + z * b2.Y + origin.Y;
double zTemp = x * b0.Z + y * b1.Z + z * b2.Z + origin.Z;
return new XYZ (xTemp, yTemp, zTemp);
}

```

Geometry.Transform 类的属性和方法于以下内容中说明。

(1) 单位矩阵 (Identity)。变换单位矩阵 (恒等变换):



$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

(2) CreateReflection() 镜像指定的平面。如图 3-54 所示, 一面墙依参照平面被镜像, CreateReflection() 方法需要参照平面的几何平面信息 (代码 3-60)。

图 3-54 墙的镜像关系

#### 代码 3-60: 使用 Reflection() 方法

```

private Transform Reflect (ReferencePlane refPlane)
{
    Transform mirTrans = Transform.CreateReflection (refPlane.Plane);

    return mirTrans;
}

```

(3) CreateRotation() 和 CreateRotationAtPoint()。在 (0, 0, 0) 或其他指定点, 按指定角度绕指定轴旋转。

(4) CreateTranslation() 方法。通过指定向量变换, 给定三维空间向量数据, 创建变换如下:

$$(x \ y \ z) \rightarrow \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \end{pmatrix}$$

(5) 行列式 (Determinant)。变换行列式:

$$\begin{vmatrix} \text{BasisX.X} & \text{BasisY.X} & \text{BasisZ.X} \\ \text{BasisX.Y} & \text{BasisY.Y} & \text{BasisZ.Y} \\ \text{BasisX.Z} & \text{BasisY.Z} & \text{BasisZ.Z} \end{vmatrix}$$



- (6) HasReflection。这是个 Boolean 值, 指示转换是否产生镜像。
- (7) 比例 (Scale)。一个表示变换比例的值。
- (8) 逆变换 (Inverse)。一个逆变换。如果存在变换矩阵  $B$ , 其满足  $A \times B = B \times A = I$  (单位矩阵), 则变换矩阵  $A$  是可逆的。
- (9) IsIdentity。Boolean 值, 该值指示该变换是否是恒等变换。
- (10) IsTranslation。Boolean 值, 该值指示该变换是否是平移变换。Geometry.Transform 提供了执行基本矩阵运算的方法。
- (11) 相乘 (Multiply)。指定变换与变换相乘并返回结果。
- (12) ScaleBasis。缩放基向量并返回结果。
- (13) ScaleBasisAndOrigin。缩放基向量和变换原点并返回结果。
- (14) OfPoint。适用于点变换。使用 Origin 属性。
- (15) OfVector。适用于向量变换。不使用 Origin 属性。
- (16) 大致相等 (AlmostEqual)。对两个变换进行比较。“大致相等”符合 Revit 核心代码的计算机制和精度。此外, Transform 类不执行 Equal 和 “=” 运算符。

API 提供了一些快捷方式来完成几何变换, 它使用某些几何类中的 Transformed 属性, 见表 3-20。

表 3-20 变 换 方 法

类 名 称	功能描述
Curve.get_Transformed (Transform transform)	适用于指定的曲线变换
GeometryElement.GetTransformed (Transform transform)	变换原始图元的几何图形副本
Profile.get_Transformed (Transform transform)	变换轮廓并返回结果
Mesh.get_Transformed (Transform transform)	变换网格并返回结果

注: 变换方法克隆本身然后返回克隆变换的结果。

除了这些方法, 实例类 (它是图元的父类, 如族实例、链接实例、导入的 CAD 内容) 有两个方法提供对给定实例的变换。GetTransform() 方法依据如何放置实例来获取对实例的基本变换, 而 GetTotalTransform() 提供了对实例 (如导入实例) 作真北方向变换的修正变换。

## 2. 参照 (Reference)

参照对图元创建是非常有用的。

- 尺寸创建需要参照。
- 参照以灵活的方式识别几何表示树路径。
- 树用于创建视图专有的几何表示。

基于 Pick 指针类型的不同, API 公开了四种参照类型。它们以不同的方式从 API 中检索: 对于点——Curve.EndPointReference 属性。

对于曲线 (直线、圆弧等)——Curve.Reference 属性。

对于面——Face.Reference 属性。



对于切边——Edge.Reference 属性。

不同参照类型不能任意使用。例如：

NewLineBoundaryConditions()方法需要对直线的参照。

NewAreaBoundaryConditions()方法需要对面的参照。

NewPointBoundaryConditions()方法需要对点的参照。

Reference. ConvertToStableRepresentation()方法可用于以字符串存储对几何对象的参照，如面、边或曲线。以后可在同一 Revit 会话中（甚至在同一文件中的不同会话中），以该字符串作为输入，使用 ParseFromStableRepresentation()方法来获取完全相同的 Reference。

(1) 选项 (Options)。几何通常是从索引属性 Element.Geometry 提取。梁、柱或支撑的原始几何对象，在实例进行连接，切割、复制、拉伸或其他后处理作改动之前，均可用 FamilyInstance.GetOriginalGeometry() 方法提取。Element.Geometry() 和 FamilyInstance.GetOriginalGeometry() 两者都接受一个给定的选项类。该选项类根据其属性定制得到的输出类型：

- ComputeReferences: 检索几何信息时，指示是否求出几何参照。默认值为 false，因此，若此属性未设置为 true，参照是不可访问的。
- IncludeNonVisibleObjects: 表示返回结果还包括默认视图中不可见的几何对象。
- View: 从指定视图中获取几何信息。请注意，若某视图被指定，则将使用该视图的详细程度替代 “DetailLevel”。
- DetailLevel: 表示建议的详细程度。默认为 “Medium”。

(2) 计算参照 (ComputeReferences)。如果此属性设置为 false，则 API 不计算几何参照。从几何树中检索的所有 Reference 属性不会返回任何内容。有关参照的详细信息，请参阅 3.7.2 节中的参照。当使用 FamilyInstance.GetOriginalGeometry() 时，此选项不可设置为 true。

(3) 包括不可见对象 (IncludeNonVisibleObjects)。大多数不可见的几何对象为用户编辑图元时所看到的构造和条件几何对象（如窗族实例的中心平面）。此属性的默认值为 false。然而，某些“条件可见几何”可能表示真实具体的对象，如 Revit MEP 中环绕风管的隔热层，则也应该被提取出来。

(4) 视图 (View)。如果用户设置不同视图的 View 属性，所检索的几何信息也会不同。查看以下示例可获得更多信息。

在 Revit 三维视图中，绘制一个楼梯然后选择裁剪区域，裁剪区域可见，Section Box 属性也显示在三维视图中。在裁剪区域内，修改三维视图中剖面框以显示部分楼梯。若使用 API 并设置该三维视图作为 Options.View 属性，来获取楼梯的几何信息，则只能检索到一部分楼梯。图 3-55 显示了 Revit 应用程序中的楼梯和用 API 绘制的楼梯。

在 Revit 中画一个楼梯然后画一个剖面如图 3-56 所示。若使用 API 并设置此剖面视图作为 Options.View 属性，来获取该楼梯的几何信息，则只能检索到一部分楼梯。用 API 绘制的楼梯见图 3-56。

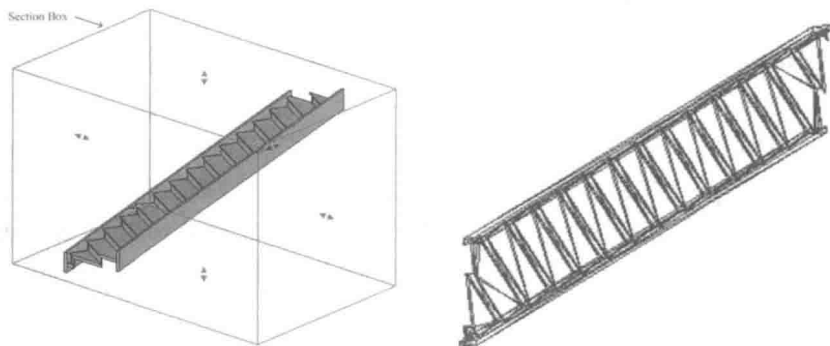


图 3-55 不同的剖面框显示不同的几何图形

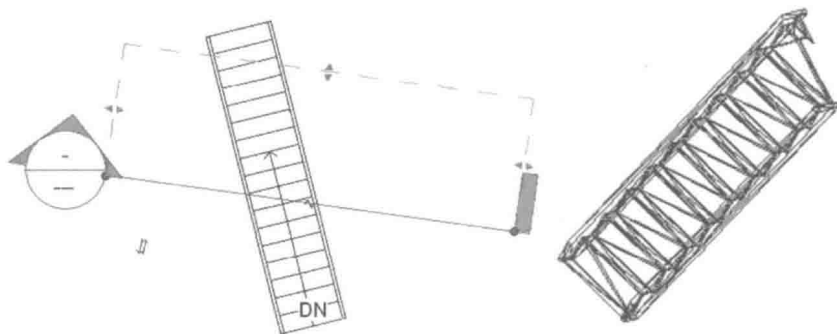


图 3-56 检索几何图形剖面视图

(5) 详细程度 (DetailLevel)。API 在 Geometry. Options. DetailLevels 中定义了三个枚举。这三个枚举对应于 Revit 应用程序中的三种详细程度, 见图 3-57。

根据 DetailLevel 属性的不同设置, 检索到的几何信息也不同。例如, 在 Revit 应用程序中绘制一根梁, 然后用 API 获取梁的几何信息并绘制。图 3-58 显示了绘制结果。

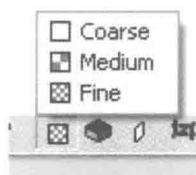


图 3-57 三种详细程度

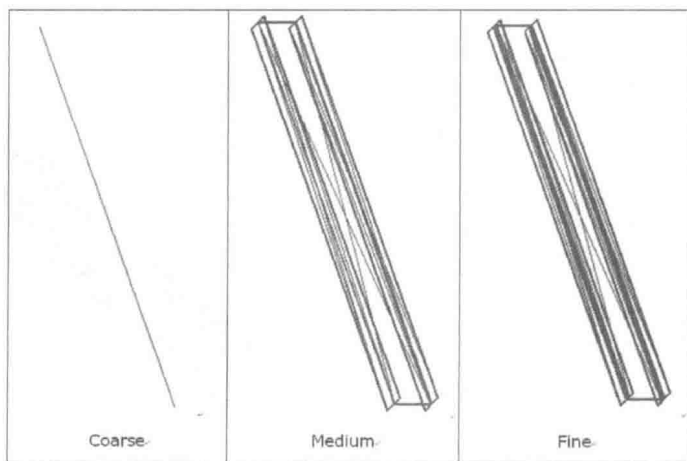


图 3-58 梁的详细几何形状



### 3. 三维边界框 (BoundingBoxXYZ)

BoundingBoxXYZ 定义了与各坐标轴平行的三维矩形框。类似于实例类, BoundingBoxXYZ 存储的也是局部坐标空间数据。它有一个 Transform 属性, 可将框局部坐标空间数据转换为模型空间数据。也就是说, 要获取模型空间中 (与 Revit 中相同) 框的边界, 必须用 Transform 属性转换每个数据成员。以下内容说明了如何使用 BoundingBoxXYZ。

(1) 定义视图边界 (Define the View Boundaries)。通过 View.CropBox 属性, BoundingBoxXYZ 可用于定义视图边界。图 3-59 使用剖面视图来说明 Revit 应用程序如何使用 BoundingBoxXYZ。

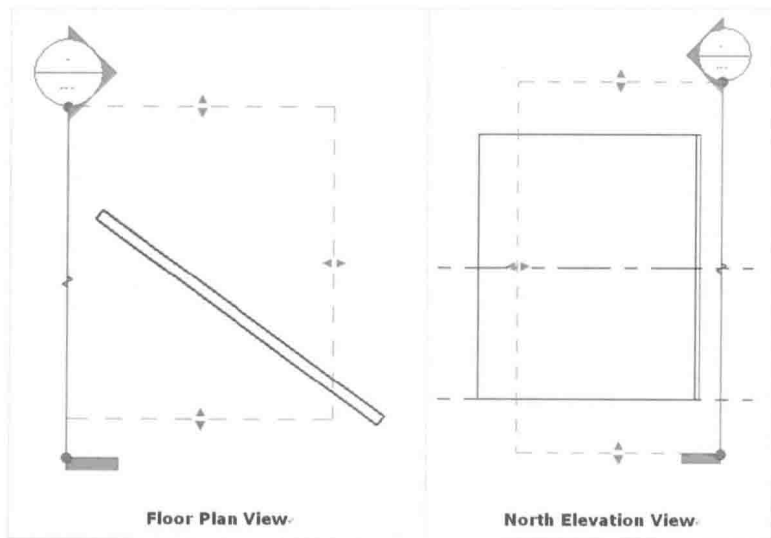


图 3-59 剖面视图中的三维边界框

图 3-59 中的虚线表示作为 CropBox 属性 (BoundingBoxXYZ 实例) 显示的剖面视图边界。图 3-60 显示了相应的剖面视图, 视图边界以外的墙段未显示。

(2) 定义剖面框 (Define a Section Box)。三维边界框也可用来定义三维视图从 View3D.SectionBox 属性检索的剖面框。在属性对话框中选择剖面框属性即可得到剖面框, 见图 3-61。

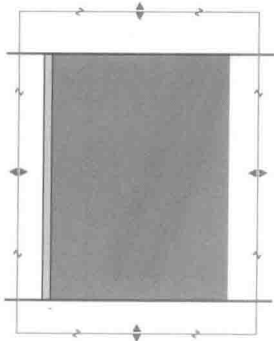


图 3-60 创建剖面视图

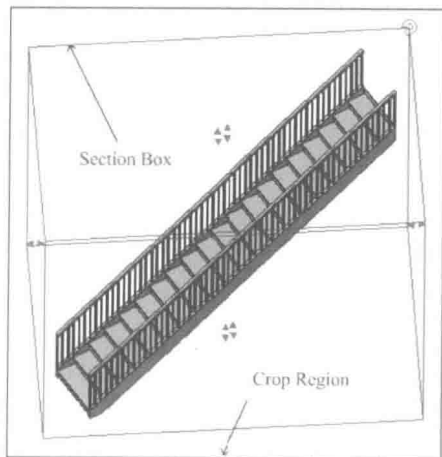


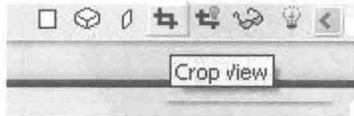

图 3-61 三维视图剖面框

### (3) 其他用途 (Other Uses)。

- 定义一个包围图元几何图形的边界框 (Element. BoundingBox 属性)。以这种方式检索的 BoundingBoxXYZ 实例是平行于坐标轴的。
- 用于 ViewSection.CreateDetail() 方法。

表 3-21 列出了该类的主要用途。

表 3-21 BoundingBoxXYZ 属性

属性名称	用 途
Max/Min	最大/最小坐标。这两个属性定义平行于各坐标轴的三维框。Transform 属性提供一个转换矩阵，将边界框变换到指定位置
Transform	将框坐标空间变换到模型空间
Enabled	指示是否启用边界框
MaxEnabled/ MinEnabled	定义给定尺寸的最大/最小边界是否处于活动状态。若“启用”属性为 false，这两个属性亦返回 false。
	<div>  <p>若裁剪视图已启用，MaxEnabled 属性和 MinEnabled 属性均返回 true</p> </div> <div>  <p>若裁剪视图未启用，MaxEnabled 属性和 MinEnabled 属性均返回 false</p> </div> <p>此属性指示是否可以使用视图裁剪框的表面裁剪图元的视图。 如果 BoundingBoxXYZ 是由 View3D.SectionBox 属性检索的，则返回值取决于是否在三维视图属性对话框中选择了 Section Box 属性。若已选择，则所有 Enabled 属性都返回 true。 若 BoundingBoxXYZ 是由 Element.BoundingBox 属性检索的，则所有 Enabled 属性为 true</p>
Bounds	Max/Min 属性封装器
BoundEnabled	MaxEnabled/MinEnabled 属性封装器

代码 3-61 阐释了如何旋转 BoundingBoxXYZ 来更改三维视图剖面框。

代码 3-61：旋转三维边界框

```
private void RotateBoundingBox (View3D view3d)
{
    BoundingBoxXYZ box = view3d.GetSectionBox();
    if (false == box.Enabled)
    {
        TaskDialog.Show ("Revit", "The section box for View3D isn't Enable.");
        return;
    }
    // Create a rotation transform,
    XYZ origin = new XYZ (0, 0, 0);
    XYZ axis = new XYZ (0, 0, 1);
```





```
Transform rotate = Transform.CreateRotationAtPoint (axis, 2, origin);  
// Transform the View3D's GetSectionBox( ) with the rotation transform  
box.Transform = box.Transform.Multiply (rotate);  
view3d.SetSectionBox (box);  
}
```

#### 4. UV 边界框 (BoundingBoxUV)

UV 边界框是一个值类，定义了平行于二维坐标轴的矩形。它具有 Min 和 Max 数据成员。它们共同定义 UV 边界框的边界。BoundingBoxUV 由 View.Outline 属性检索，是图纸空间视图中的视图边界，见图 3-62。

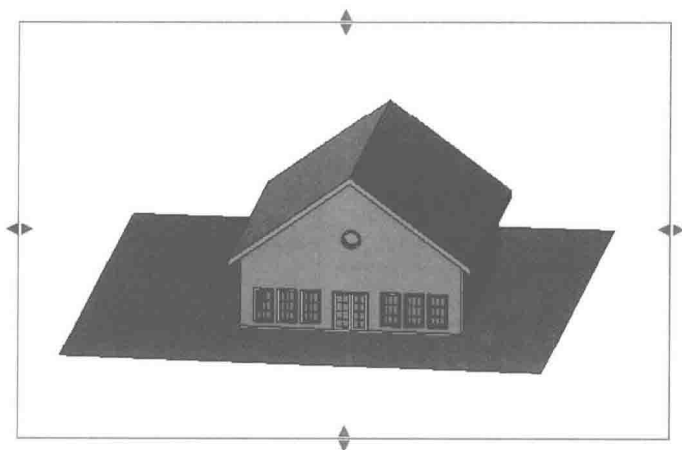


图 3-62 视图轮廓

Min 点、Max 点两点定义一个 BoundingBoxUV (图 3-63)。

- Min 点：左下角端点。
- Max 点：右上角端点。

注意：UV 边界框不能表示如图 3-64 所示的倾斜矩形。

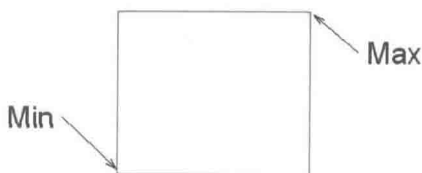


图 3-63 UV 边界框的最大和最小点

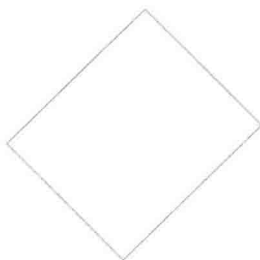


图 3-64 倾斜矩形

### 3.7.4 集合类 (Collection Classes)

基于集合类所包含的项目，表 3-22 列出了 API 提供的集合类。



表 3-22 几何的集合类

类/类型	对应的集合类	对应的迭代器
Edge	EdgeArray, EdgeArrayArray	EdgeArrayIterator, EdgeArrayArrayIterator
Face	FaceArray	FaceArrayIterator
GeometryObject	GeometryObjectArray	GeometryObjectArrayIterator
Instance	InstanceArray	InstanceArrayIterator
Mesh	MeshArray	MeshArrayIterator
Reference	ReferenceArray	ReferenceArrayIterator
Solid	SolidArray	SolidArrayIterator
Double value	DoubleArray	DoubleArrayIterator

所有这些类使用非常类似的方法和属性来完成类似的任务。有关详细信息, 请参阅 2.4 节集合。

### 3.7.5 示例: 检索梁的几何数据 (Example: Retrieve Geometry Data from a Beam)

本节说明了如何获取梁的体和曲线, 亦可以用类似的方式检索柱和支撑的几何数据。GeometryElement 可能包含所需的诸如 Solid 或 GeometryInstance, 这取决于该梁是与其他图元连接的还是独立的, 代码 3-62 涵盖了这两种情况。

注意: 如还想找到梁和支撑的驱动曲线, 则调用 FamilyInstance Location 属性可获得 LocationCurve。

代码 3-62: 获取梁的体和曲线

```
public void GetCurvesFromABeam (Autodesk.Revit.DB.FamilyInstance beam, Autodesk.Revit.DB.Options options)
{
    Autodesk.Revit.DB.GeometryElement geomElem = beam.get_Geometry (options);

    Autodesk.Revit.DB.CurveArray curves = new CurveArray ();
    Autodesk.Revit.DB.SolidArray solids = new SolidArray ();

    //Find all solids and insert them into solid array
    AddCurvesAndSolids (geomElem, ref curves, ref solids);
}

private void AddCurvesAndSolids (Autodesk.Revit.DB.GeometryElement geomElem,
    ref Autodesk.Revit.DB.CurveArray curves,
    ref Autodesk.Revit.DB.SolidArray solids)
{
    foreach (Autodesk.Revit.DB.GeometryObject geomObj in geomElem)
    {
        Autodesk.Revit.DB.Curve curve = geomObj as Autodesk.Revit.DB.Curve;
        if (null != curve)
```



```
{
    curves.Append (curve);
    continue;
}
Autodesk.Revit.DB.Solid solid = geomObj as Autodesk.Revit.DB.Solid;
if (null != solid)
{
    solids.Append (solid);
    continue;
}
//If this GeometryObject is Instance, call AddCurvesAndSolids
Autodesk.Revit.DB.GeometryInstance geomInst = geomObj as Autodesk.Revit.DB.GeometryInstance;
if (null != geomInst)
{
    Autodesk.Revit.DB.GeometryElement transformedGeomElem = geomInst.GetInstanceGeometry
(geomInst.Transform);
    AddCurvesAndSolids (transformedGeomElem, ref curves, ref solids);
}
}
```

上述示例使用 `FamilyInstance.Geometry` 属性来访问梁的实际几何形状。要获取族实例被连接、切割、复制、拉伸或其他后处理作更改之前的原始几何形状,请使用 `FamilyInstance.GetOriginalGeometry()` 方法。

注:有关如何检索 `Geometry.Options` 类型对象的更多信息,请参阅 3.7.3 节中的 `Geometry.Options`。

### 3.7.6 体拉伸分析 (Extrusion Analysis of a Solid)

实用工具类 `ExtrusionAnalyzer` 尝试将给定的一片几何形状调整为拉伸轮廓的形状。此类的实例是个单次用途的类,它需要提供一个体几何、一个平面和一个方向。在 `ExtrusionAnalyzer` 初始化后,通过下列成员可访问其结果:

- `GetExtrusionBase()` 方法返回求得的与输入平面对齐的拉伸体基本轮廓。
- `CalculateFaceAlignment()` 方法用于确定原始几何模型的所有面是否都与计算的拉伸面对齐。在某些情况下是非常有用的,例如,如果墙顶有个倾斜的连接面,则可估计到墙顶可能已与屋顶连接。若某个面未对齐,则说明有某个影响到该面的图元连接到此几何形状。
- 要确定产生不对齐面的图元,将该面传递到 `Element.GetGeneratingElementIds()`。有关此实用程序的详细信息,请看下节。

`ExtrusionAnalyzer` 实用程序最适合用于那些至少有点“类似拉伸”的几何体,例如,墙几何体也许会也许不会受到端部连接、楼板连接、屋顶连接、门窗切割形成洞口或其他改动的影响。而极少用于那些特殊形状和方向组合情况,分析器可能无法确定合适的面作为拉伸的基面,此时将引发非法操作的异常。



在代码 3-63 中，拉伸分析器用于计算并显示输入体在光线方向上所形成的阴影。

**代码 3-63：使用拉伸分析器计算并绘制阴影轮廓**

```

/// <summary>
/// Draw the shadow of the indicated solid with the sun direction specified.
/// </summary>
/// <remarks>The shadow will be outlined with model curves added to the document.
/// A transaction must be open in the document.</remarks>
/// <param name="document">The document.</param>
/// <param name="solid">The target solid.</param>
/// <param name="targetLevel">The target level where to measure and
/// draw the shadow.</param>
/// <param name="sunDirection">The direction from the sun (or light source) .</param>
/// <returns>The curves created for the shadow.</returns>
/// <throws cref="Autodesk.Revit.Exceptions.InvalidOperationException">
/// Thrown by ExtrusionAnalyzer when the geometry and
/// direction combined do not permit a successful analysis.</throws>
private static ICollection<ElementId> DrawShadow (Document document, Solid solid,
                                                Level targetLevel, XYZ sunDirection)
{
    // Create target plane from level.
    Plane plane = document.Application.Create.NewPlane (XYZ.BasisZ,
                                                        new XYZ (0, 0, targetLevel.ProjectElevation));

    // Create extrusion analyzer.
    ExtrusionAnalyzer analyzer = ExtrusionAnalyzer.Create (solid, plane, sunDirection);

    // Get the resulting face at the base of the calculated extrusion.
    Face result = analyzer.GetExtrusionBase();

    // Convert edges of the face to curves.
    CurveArray curves = document.Application.Create.NewCurveArray();
    foreach (EdgeArray edgeLoop in result.EdgeLoops)
    {
        foreach (Edge edge in edgeLoop)
        {
            curves.Append (edge.AsCurve());
        }
    }

    // Get the model curve factory object.
    Autodesk.Revit.Creation.ItemFactoryBase itemFactory;
    if (document.IsFamilyDocument)
        itemFactory = document.FamilyCreate;
    else
        itemFactory = document.Create;

    // Add a sketch plane for the curves.
    SketchPlane sketchPlane =

```



```
itemFactory.NewSketchPlane (document.Application.Create.NewPlane (curves));
document.Regenerate();

// Add the shadow curves
ModelCurveArray curveElements = itemFactory.NewModelCurveArray (curves, sketchPlane);

// Return the ids of the curves created
List<ElementId> curveElementIds = new List<ElementId>();
foreach (ModelCurve curveElement in curveElements)
{
    curveElementIds.Add (curveElement.Id);
}

return curveElementIds;
}
```

代码 3-63 可用于在当前日光和视图阴影设置条件下, 计算给定体量的阴影 (图 3-65)。

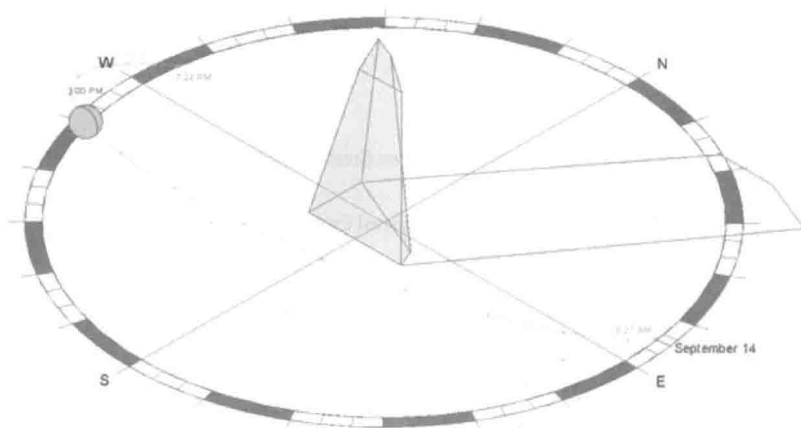


图 3-65 在当前日光和视图阴影设置条件下计算给定体量的阴影

### 3.7.7 由光线投影找出几何体 (Finding Geometry by Ray Projection)

参照求交器 (ReferenceIntersector) 类允许应用程序使用 Revit 选择工具来找到图元和几何体。该类使用来自指定方向上的点的光线来查找由光线命中的几何体。

此类只与三维几何体相交, 需要创建一个三维视图。可能使用一个已由剖面框切取的三维视图, 或有视图专有几何体及图形选项设置的三维视图, 来找出未切割和未裁剪的三维模型中未发现的交集。

ReferenceIntersector 类支持根据图元或参照类型过滤输出。根据所使用的构造函数, 或在调用方法执行光线投影之前使用类的方法和属性, 可以自定义输出。

ReferenceIntersector 类有表 3-23 中的四个构造函数。

表 3-23                      ReferenceIntersector 类的四个构造函数

名 称	说 明
ReferenceIntersector ( View3D )	构造一个参照求交器, 设置为返回所有图元的交集并表示所有参照目标类型
ReferenceIntersector ( ElementFilter, FindReferenceTarget, View3D )	构造一个参照求交器, 设置为返回通过过滤器的任何图元的交集
ReferenceIntersector ( ElementId, FindReferenceTarget, View3D )	构造一个参照求交器, 设置为仅返回单个目标图元的交集
ReferenceIntersector ( ICollection<ElementId>, FindReferenceTarget, View3D )	构造一个参照求交器, 设置为返回任何一组目标图元的交集

FindReferenceTarget 枚举选项包括 Element、Mesh、Edge、Curve、Face 或其全部。

有两种方法投影光线, 两者都需要输入光源和光线方向。Find() 方法返回一个与参照求交器的条件匹配的 ReferenceWithContext 对象集合。此对象包含与光线相交的参照, 它可以是图元参照, 也可以是几何参照。返回的某些图元参照会有个相应的也被相交的几何对象 (例如, 穿过墙体洞口的光线会与墙体和洞口图元相交)。如果只关注真实物理相交, 那么应用程序应该丢弃其 Reference 为类型 Element 的所有参照。

FindNearest() 方法的行为类似于 Find() 方法, 但只返回离光源最近相交参照。

ReferenceWithContext 返回包含一个 “proximity” 参数。它是光源和相交点之间的距离。对于特定的几何分析, 应用程序可以用这个距离排除离光源太远的项。应用程序还可使用此距离来解决某些涉及模型几何体所在位置的有趣问题。

注:

- 此方法找出两个能被找到的 References, 仅返回位于光线之前的图元。
- 此方法不会返回与链接文件中几何体的交集。
- 此方法不会返回不在活动设计选项中图元的交集。

1. 查找相邻图元 ( Find Elements Near Elements )

该工具的主要用途是查找紧邻其他图元的图元, 允许应用程序用此工具作为它的 “眼睛”, 确定那些还不存在内建关系的图元之间的关系。

例如, 光线追踪能力可用于查找嵌入墙中的柱 (图 3-66)。柱和墙并不保持直接关联, 此类允许我们在墙的范围以外通过追踪光线找到潜在对象, 寻找墙和柱的交集。

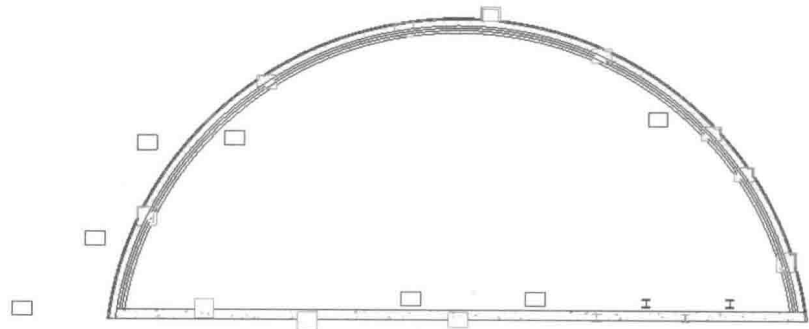


图 3-66    查找嵌入墙中的柱



## 2. 测量距离 (Measure Distances)

这个类还可用于测量天窗至最近楼层的垂直距离，见图 3-67 和代码 3-64)。

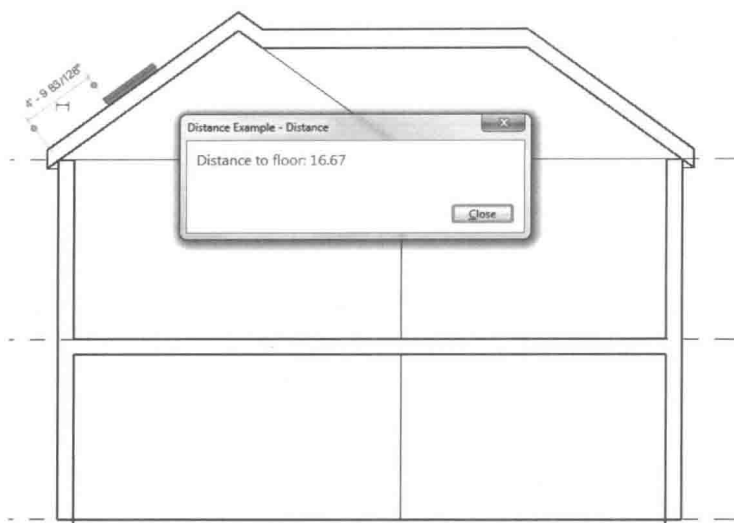


图 3-67 用 ReferenceIntersector.FindNearest() 作测量

### 代码 3-64: 用光线投影测量距离

```
public class RayProjection : IExternalCommand
{
    public Result Execute (ExternalCommandData revit, ref string message, ElementSet elements)
    {
        Document doc = revit.Application.ActiveUIDocument.Document;

        Selection selection = revit.Application.ActiveUIDocument.Selection;

        // If skylight is selected, process it.
        FamilyInstance skylight = null;
        if (selection.Elements.Size == 1)
        {
            foreach (Element e in selection.Elements)
            {
                if (e is FamilyInstance)
                {
                    FamilyInstance instance = e as FamilyInstance;
                    bool isWindow = (instance.Category.Id.IntegerValue == (int) BuiltInCategory.OST_Windows);
                    bool isHostedByRoof = (instance.Host.Category.Id.IntegerValue == (int) BuiltInCategory.OST_Roofs);

                    if (isWindow && isHostedByRoof)
                    {
                        skylight = instance;
                    }
                }
            }
        }
    }
}
```



```

    }
}

if (skylight == null)
{
    message = "Please select one skylight.";
    return Result.Cancelled;
}

// Calculate the height
Line line = CalculateLineAboveFloor (doc, skylight);

// Create a model curve to show the distance
Plane plane = revit.Application.Create.NewPlane (new XYZ (1, 0, 0), line.GetEndPoint (0));
SketchPlane sketchPlane = SketchPlane.Create (doc, plane);

ModelCurve curve = doc.Create.NewModelCurve (line, sketchPlane);

// Show a message with the length value
TaskDialog.Show ("Distance", "Distance to floor: " + String.Format ("{0: f2}", line.Length));

return Result.Succeeded;
}

/// <summary>
/// Determines the line segment that connects the skylight to the nearest floor.
/// </summary>
/// <returns>The line segment.</returns>
private Line CalculateLineAboveFloor (Document doc, FamilyInstance skylight)
{
    // Find a 3D view to use for the ReferenceIntersector constructor
    FilteredElementCollector collector = new FilteredElementCollector (doc);
    Func<View3D, bool> isNotTemplate = v3 => ! (v3.IsTemplate);
    View3D view3D = collector.OfClass (typeof (View3D)) .Cast<View3D>().First<View3D> (isNotTemplate);

    // Use the center of the skylight bounding box as the start point.
    BoundingBoxXYZ box = skylight.get_BoundingBox (view3D);
    XYZ center = box.Min.Add (box.Max) .Multiply (0.5);

    // Project in the negative Z direction down to the floor.
    XYZ rayDirection = new XYZ (0, 0, -1);

    ElementClassFilter filter = new ElementClassFilter (typeof (Floor));

    ReferenceIntersector refIntersector = new ReferenceIntersector (filter, FindReferenceTarget.Face, view3D);
    ReferenceWithContext referenceWithContext = refIntersector.FindNearest (center, rayDirection);

    Reference reference = referenceWithContext.GetReference();
    XYZ intersection = reference.GlobalPoint;

```





```
// Create line segment from the start point and intersection point.
Line result = Line.CreateBound (center, intersection);
return result;
```

### 3. 光线弹射/分析 (Ray Bouncing/Analysis)

ReferenceIntersector.Find()返回的参照包括几何上的交点。已知面上的交点、面的材质和光线方向,可让应用程序分析建筑物内的反射和折射。图 3-68 演示了如何使用交点来反射与模型图元相交的光线,并添加表示每条光线路径的模型曲线。

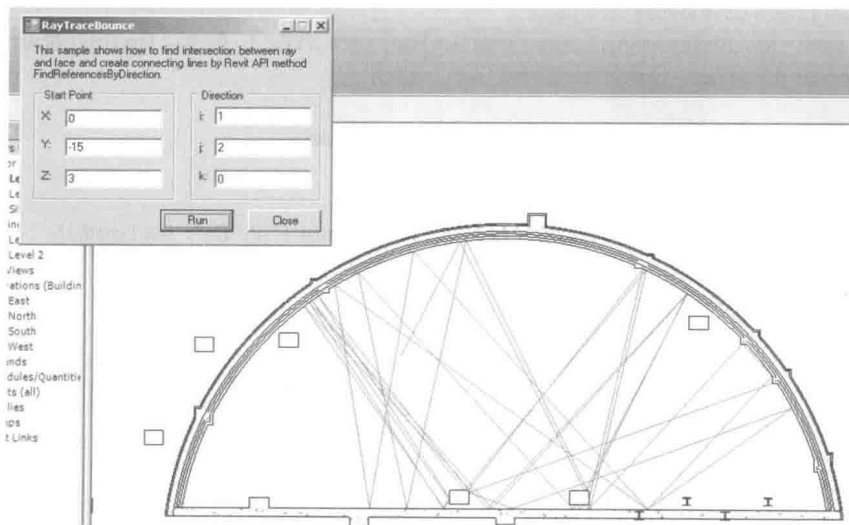


图 3-68 光线反射相交面

### 4. 查找交叉/碰撞 (Find Intersections/Collisions)

ReferenceIntersector 类的另一个运用是碰撞检查,(如梁或管道)与给定梁或管道中心线的交叉/干扰(图 3-69)。

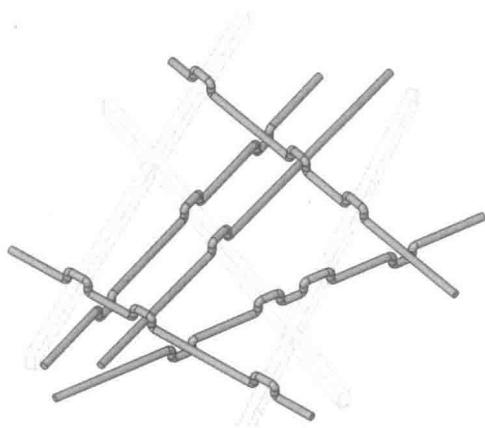


图 3-69 绕过干扰重排图元

### 3.7.8 几何实用程序类 (Geometry Utility Classes)

#### 1. 主体对象实用程序 (HostObjectUtils)

HostObjectUtils 类提供作为快捷方式的方法,来定位复合主体对象的某些面。这些实用程序检索那些作为对象的 CompoundStructure 边界的面。

- HostObjectUtils.GetSideFaces(): 适用于

Walls 和 FaceWalls, 可获取外部或内部装饰面。



- `HostObjectUtils.GetTopFaces()` 和 `HostObjectUtils.GetBottomFaces()`: 适用于屋顶、楼板和天花板。

## 2. 体实用程序 (SolidUtils)

`SolidUtils` 类包含对体执行操作的方法。

- `SolidUtils.SplitVolumes()`: 需要一个其中包含分离封闭体积的体, 返回代表各自体积的新创建 `Solid` 对象。如果不必分离, 那么返回原输入体。
- `SolidUtils.TessellateSolidOrShell()`: 把给定的输入 `Solid` (它可以是一个或多个完全封闭的体积, 或敞开的壳体) 三角化。返回一个 `TriangulatedSolidOrShell` 对象, 允许访问所存储体的三角化边界组件或壳体的三角化连接组件。

## 3. 几何连接实用程序 (JoinGeometryUtils)

`JoinGeometryUtils` 类包含一些连接和拆解图元的方法, 并用于管理图元的连接顺序。这些实用程序不适用于族文件。

- `JoinGeometryUtils.AreElementsJoined()`: 确定两个图元是否已连接。
- `JoinGeometryUtils.GetJoinedElements()`: 返回已连接到指定图元的所有图元。
- `JoinGeometryUtils.JoinGeometry()`: 在有共享公共面的两个图元之间创建连接, 并删除连接图元之间的可见边。连接后的图元共享相同的线宽和填充图案。
- `JoinGeometryUtils.UnjoinGeometry()`: 删除两个连接图元之间的连接。
- `JoinGeometryUtils.SwitchJoinOrder()`: 颠倒两个图元的连接顺序。剪切图元变成被剪切图元, 反之亦然。
- `JoinGeometryUtils.IsCuttingElementInJoin()`: 确定两个连接图元中, 是否为第一个图元剪切第二个图元, 或第二个图元剪切第一个图元。

## 4. 细小面应用程序 (FacetingUtils)

此类用于将三角化结构转换为其中一些三角形已合并成四边形的结构。

- `FacetingUtils.ConvertTrianglesToQuads()`: 以 `TriangulationInterface` (由 `TriangulatedSolidOrShell` 构建) 作为输入, 返回一个三角形和四边形的集合来表示原始三角化对象。

### 3.7.9 房间和空间几何对象 (Room and Space Geometry)

Revit API 提供了对三维空间图元几何体 (房间和空间) 的访问。

`SpatialElementGeometryCalculator` 类可用于计算空间图元几何体, 并获取几何体和图元的边界图元之间的关系。可为此实用程序提供两个选项:

- `SpatialElementBoundaryLocation`: 是否使用装饰面或边界图元中心线作计算。
- `StoredFreeBoundaryFaces`: 结果中是否要包括那些不直接映射到边界图元的面。
- 几何计算结果包含在类 `SpatialElementGeometryResults` 中。可从该类获得:
- 表示几何体的体体积 [`GetGeometry()` 方法]。
- 边界面信息 (一个 `SpatialElementBoundarySubfaces` 集合)。

每个子面提供:

- 空间图元的面。



- 边界图元的匹配面。
- 子面 (以此特定边界图元为边界的空间图元面的一部分)。
- 子面类型 (底面、顶面、侧面)。

有关使用此实用程序的一些说明如下:

- 该计算器会为已处理的几何体保持内部缓存。如果计算同一项目中几个图元的几何体, 则应使用此类的单个实例。要注意的是, 当对文件作任何更改时, 缓存将被清除。
- 楼板几乎从不作为边界图元考虑。Revit 使用房间的二维轮廓形成底面, 并不将它们与楼板几何体匹配。
- 由“墙-切割”特性创建的开口如门窗, 不包括在返回的面中。
- 几何计算与 Revit 所提供的能力相匹配。某些情况下, Revit 会作出如何计算房间和空间的界限体积的假设, 这些假设会出现在实用程序的输出中。

代码 3-65 示例了如何找出房间的边界面并计算面积。

**代码 3-65: 用 SpatialElementGeometryCalculator 计算面的面积**

```
SpatialElementGeometryCalculator calculator = new SpatialElementGeometryCalculator (doc);

// compute the room geometry
SpatialElementGeometryResults results = calculator.CalculateSpatialElementGeometry (room);

// get the solid representing the room's geometry
Solid roomSolid = results.GetGeometry( );

foreach (Face face in roomSolid.Faces)
{
    double faceArea = face.Area;

    // get the sub-faces for the face of the room
    IList<SpatialElementBoundarySubface> subfaceList = results.GetBoundaryFaceInfo (face);
    foreach (SpatialElementBoundarySubface subface in subfaceList)
    {
        if (subfaceList.Count > 1) // there are multiple sub-faces that define the face
        {
            // get the area of each sub-face
            double subfaceArea = subface.GetSubface( ).Area;

            // sub-faces exist in situations such as when a room-bounding wall has been
            // horizontally split and the faces of each split wall combine to create the
            // entire face of the room
        }
    }
}
```

代码 3-66 示例了如何计算房间几何, 并找出其属于定义该房间的图元面所用的材料。



代码 3-66: 用 SpatialElementGeometryCalculator 找出面的材料

```

public void MaterialFromFace()
{
    string s = "";
    Document doc = this.Document;
    UIDocument uidoc = new UIDocument ( doc );
    Room room = doc.GetElement ( uidoc.Selection.PickObject ( ObjectType.Element ) .ElementId ) as Room;

    SpatialElementBoundaryOptions spatialElementBoundaryOptions = new SpatialElementBoundaryOptions();
    spatialElementBoundaryOptions.SpatialElementBoundaryLocation = SpatialElementBoundaryLocation.Finish;
    SpatialElementGeometryCalculator calculator = new SpatialElementGeometryCalculator ( doc, spatialElementBoundaryOptions );
    SpatialElementGeometryResults results = calculator.CalculateSpatialElementGeometry ( room );
    Solid roomSolid = results.GetGeometry();

    foreach ( Face roomSolidFace in roomSolid.Faces )
    {
        foreach ( SpatialElementBoundarySubface subface in results.GetBoundaryFaceInfo ( roomSolidFace ) )
        {
            Face boundingElementface = subface.GetBoundingElementFace();
            ElementId id = boundingElementface.MaterialElementId;
            s += doc.GetElement ( id ).Name + ", id = " + id.IntegerValue.ToString() + "\n";
        }
    }
    TaskDialog.Show ( "revit", s );
}

```

## 3.8 草图 (Sketching)

要在 Revit 中创建图元或编辑它们的轮廓, 需要首先创建草图对象。需要草图的图元示例包括:

- Roofs (屋顶)。
- Floors (楼板)。
- Stairs (楼梯)。
- Railings (扶手)。

也需要草图来定义的其他类型的几何图形, 诸如:

- Extrusions (拉伸)。
- Openings (洞口)。
- Regions (区域)。

Revit 平台 API 中, 草图功能由二维和三维草图类表示。

二维草图:

- SketchPlane (草图平面)。
- Sketch (草图)。



- ModelCurve (模型曲线)。
- 其他, 等等。

三维草图:

- GenericForm (一般形式)。
- Path3D (三维路径)。

除了草图图元以外, 本节还介绍了模型曲线。图元分类的详细信息, 请参见 1.5.1 节图元分类。

### 3.8.1 二维草图 (The 2D Sketch Class)

Sketch 类表示用于创建三维模型的平面内的封闭曲线。其最重要的特性由 SketchPlane 和 CurveLoop 属性表示。

编辑 Revit 文件时, 不能通过迭代 Document.Elements 枚举来检索 Sketch 对象, 因为所有 Sketch 对象都是瞬态图元。当访问族的三维建模信息时, Sketch 对象对成形几何体是非常重要的。SketchPlane 是所有二维草图如模型曲线和草图类的基础, 它还是二维注释图元如详图曲线的基础。ModelCurve 和 DetailCurve 都有 SketchPlane 属性, 且需要有与创建方法对应的 SketchPlane。在 Revit 用户界面中, 草图平面始终是不可见的。

每个模型曲线都必须位于一个草图平面内。换句话说, 无论是在用户界面中还是通过 API 绘制模型曲线, 都必须有一个草图平面。因此, 绘制模型曲线的二维视图中至少有一个草图平面存在。

二维视图包含天花板平面、楼板平面和立面视图类型。默认情况下, 所有这些视图会自动创建草图平面。与二维视图相关的草图平面名称返回该视图的名称如“标高 1”或“北立面”。

在用户界面要指定一个新的工作平面, 可以选择图 3-70 所示“Specify a New Work Plane”。在点选“Specify a New Work Plane”后, 选取特定图元上某个面如图 3-71 所示的墙面。这种情况下, SketchPlane.Name 属性返回该图元所对应的字符串。例如, 在图 3-71 中, SketchPlane.Name 属性返回与 Wall.Name 属性相同的字符串“Generic-8”。



图 3-70 选择一个平面作为新的工作平面

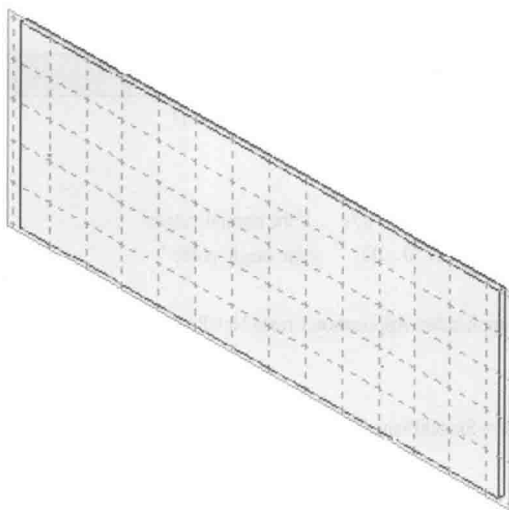


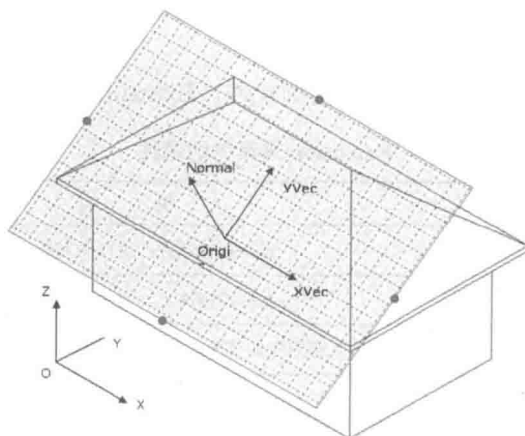
图 3-71 选择墙的一个面作为工作平面

注：草图平面不同于工作平面，工作平面是可见并可以选取的。在当前版本 API 中，它没有专属的类，但可由 `Element` 类来表示。工作平面必须根据指定的草图平面来定义。工作平面和草图平面类别属性都返回 `null`。虽然草图平面始终不可见，但总存在一个工作平面所对应的草图平面。工作平面用于在文字和图形上表达草图平面。

以下信息适用于 `SketchPlane` 成员：

- `ID`、`UniqueId`、`Name` 和 `Plane` 属性返回一个值。
- `Parameters` 属性为“空”。
- `Location` 属性返回一个位置对象。
- 其他属性返回“`null`”。

`Plane` 包含草图平面几何信息，草图平面建立了 `Plane` 平面坐标系统，见图 3-72。

图 3-72 `SketchPlane` 及其 `Plane` 坐标系统



代码 3-67 阐释了如何创建一个新的草图平面。

代码 3-67: 新建草图平面

```
private SketchPlane CreateSketchPlane (UIApplication application)
{
    //try to create a new sketch plane
    XYZ newNormal = new XYZ (1, 1, 0);    // the normal vector
    XYZ newOrigin = new XYZ (0, 0, 0);    // the origin point
    // create geometry plane
    Plane geometryPlane = application.Application.Create.NewPlane (newNormal, newOrigin);

    // create sketch plane
    SketchPlane sketchPlane = SketchPlane.Create (application.ActiveUIDocument.Document, geometryPlane);

    return sketchPlane;
}
```

### 3.8.2 三维草图 (3D Sketch)

三维草图用于编辑族或创建三维对象。在 Revit 平台 API 中, 可以使用下面的类来完成三维草图:

- Extrusion (拉伸)。
- Revolution (旋转)。
- Blend (融合)。
- Sweep (放样)。

换句话说, 可以通过以上四种操作方式将二维模型转变成三维模型。关于二维草图的更多细节, 请参阅 3.8.1 节二维草图。

#### 1. 拉伸 (Extrusion)

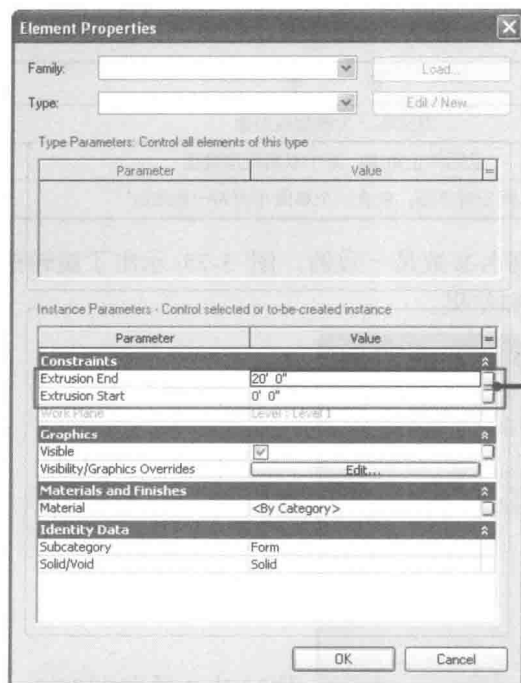
Revit 可使用拉伸来定义族中三维几何体。通过定义一个平面内的二维草图可创建拉伸, 然后 Revit 在起点和终点之间拉伸草图。

在族模型和体量中可为一般拉伸体查询拉伸。Extrusion 类具有的属性见表 3-24。

表 3-24 Extrusion 类具有的属性

属性	说 明
ExtrusionStart	返回拉伸起点, 是个双精度型数值
ExtrusionEnd	返回拉伸终点, 是个双精度型数值
Sketch	返回拉伸草图, 包含一个草图平面和一些曲线

ExtrusionStart 和 ExtrusionEnd 属性的值与 Revit 用户界面中的相应参数是一致的, 图 3-73 和图 3-74 分别示出了相应的参数和拉伸结果。



The value of ExtrusionEnd property is 20'0".  
The value of ExtrusionStart property is 0'0".

图 3-73 用户界面中的拉伸参数

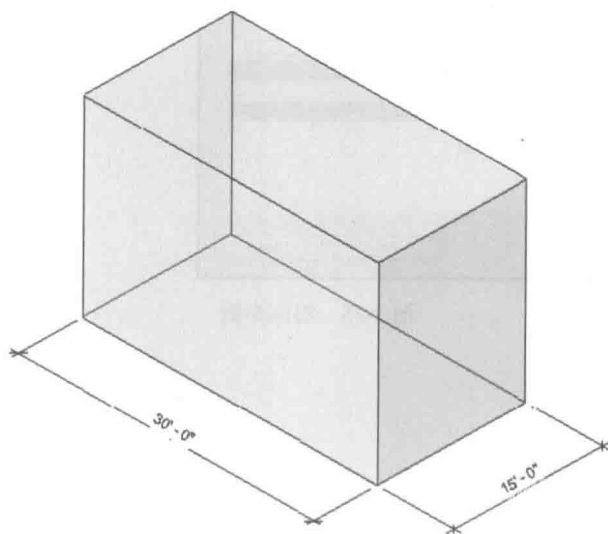


图 3-74 拉伸结果

## 2. 旋转 (Revolution)

旋转命令绕轴旋转创建几何体。门的旋钮、家具上的其他旋钮、圆屋顶或圆柱可以使用旋转命令来创建。

在族模型和体量中可为一般旋转体查询旋转对象。表 3-25 列出了 Revolution 类具有的属性。





表 3-25

Revolution 类具有的属性

属性	说 明
Axis	返回轴, 为模型线对象
EndAngle	返回终止角度, 是个双精度型数值
Sketch	返回旋转草图, 包含一个草图平面和一些曲线

终止角度与 Revit 用户界面中的同名参数是一致的, 图 3-75 示出了旋转的参数, 图 3-76 和图 3-77 分别示出了相应的草图和结果。

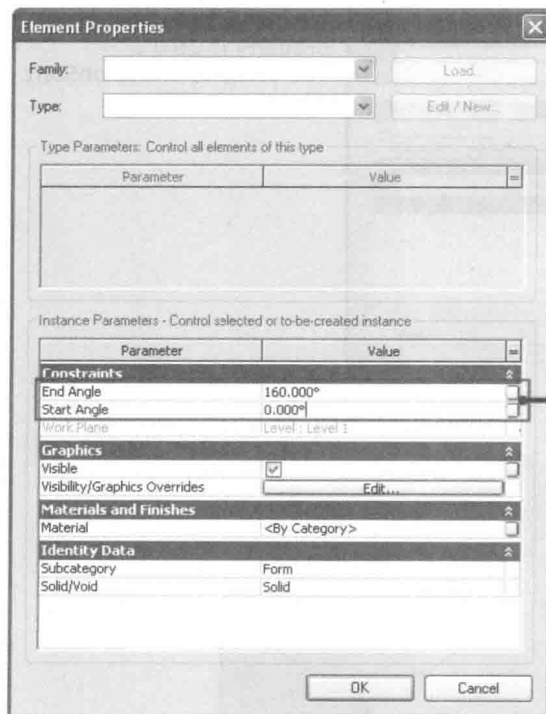


图 3-75 对应参数

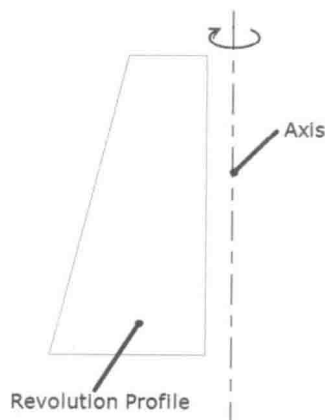


图 3-76 旋转草图

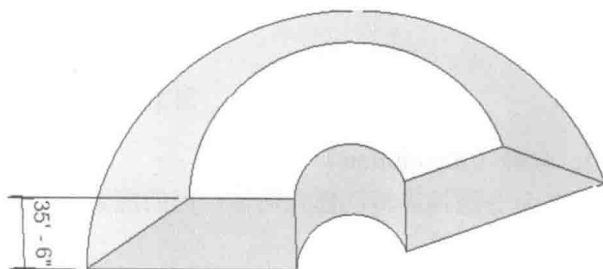


图 3-77 旋转结果

注：不可用 Revit 平台 API 来访问起始角度。若终止角度为正值，则旋转方向为顺时针向。若为负值则旋转方向为逆时针向。

3. 融合 ( Blend )

融合命令将两个轮廓融合在一起。例如，如果草绘一个大的矩形，在它顶上再画一个较小的矩形，那么 Revit Architecture 可将这两个形状融合在一起。

在族模型和体量中可为一般融合体查询融合对象。表 3-26 列出了 Blend 类具有的属性。

表 3-26 Blend 类具有的属性

属性	说 明
BottomSketch	返回底部草图，是个草图对象
TopSketch	返回顶部草图，是个草图对象
FirstEnd	返回第一个端点，是个双精度型数值
SecondEnd	返回第二个端点，是个双精度型数值

FirstEnd 和 SecondEnd 属性值都与 Revit 用户界面中的同名参数相同。图 3-78 和图 3-79 分别示出了融合所对应的参数和草图，图 3-80 为相应的融合结果。

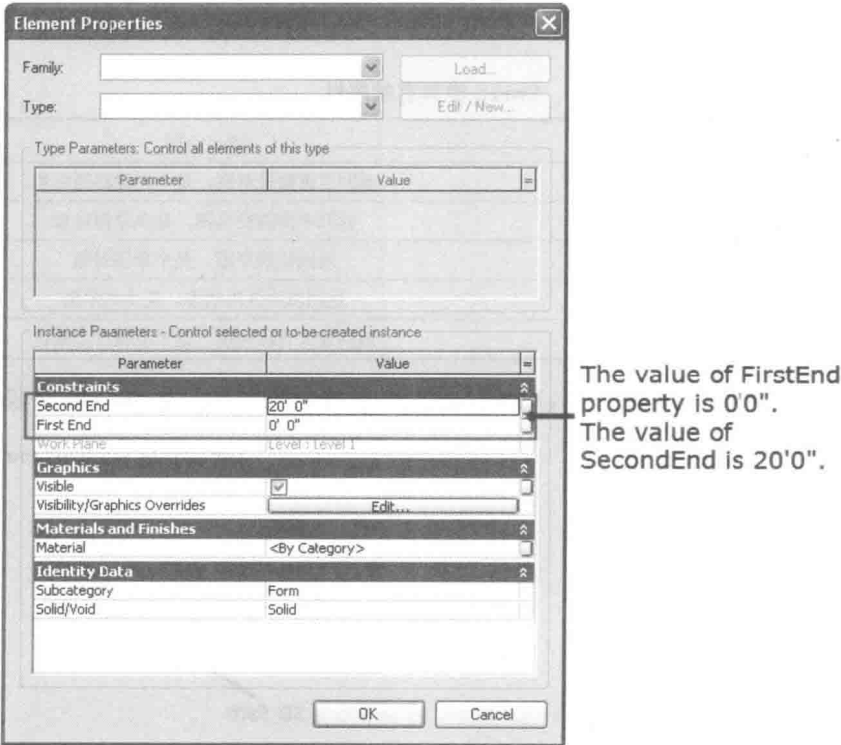


图 3-78 用户界面中的融合参数

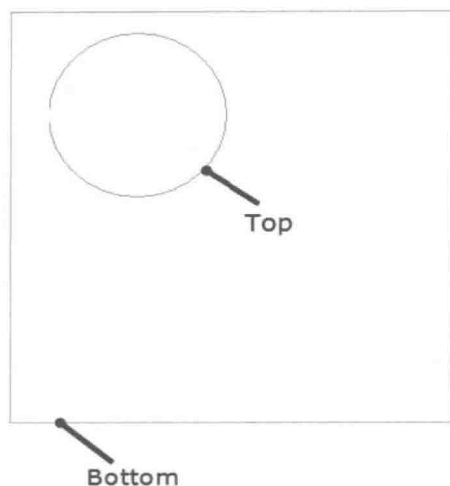


图 3-79 融合的顶部草图和底部草图

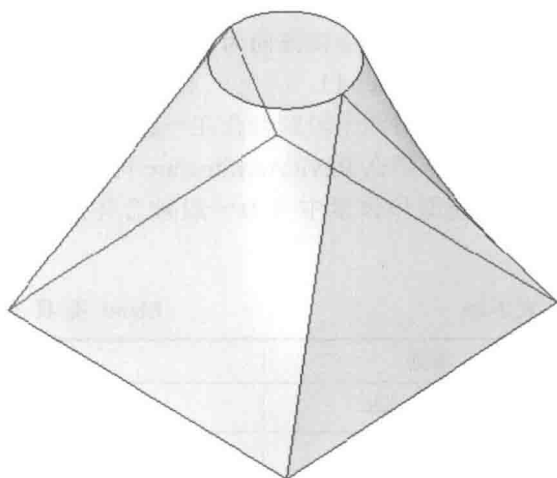


图 3-80 融合结果

#### 4. 放样 (Sweep)

放样命令沿着一条已创建的二维路径或选定的三维路径放样一个轮廓，路径可以是开放或封闭的环，但必须穿过轮廓平面，且轮廓草图与路径相垂直，见图 3-81。

在族模型和体量中可为一般放样体查询放样对象。表 3-27 列出了 Sweep 类具有的属性。

表 3-27

Sweep 类具有的属性

属性	说 明
Path3d	返回三维路径草图，是个三维路径对象
PathSketch	返回平面路径草图，是个草图对象
ProfileSketch	返回轮廓草图，是个草图对象
EnableTrajSegmentation	返回轨迹分段状态，是个布尔值
MaxSegmentAngle	返回最大分段角度，是个双精度型数值

创建二维路径与其他形状类似，三维路径通过选择已创建的三维曲线来获取(图 3-82)。

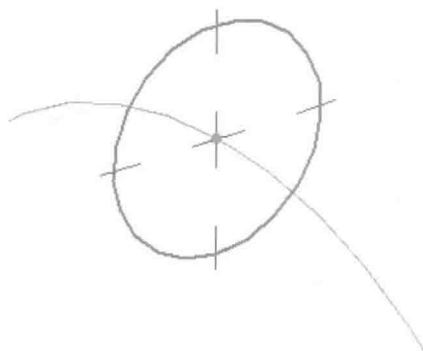


图 3-81 放样轮廓草图和路径

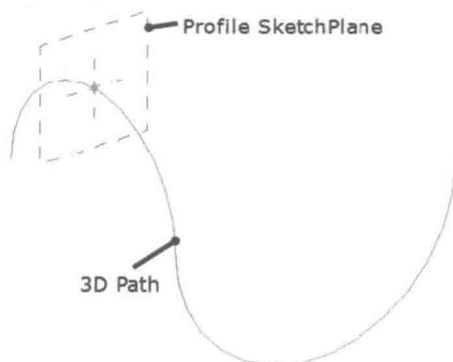


图 3-82 选择放样三维路径



注：以下信息适用于放样。

- Path3d 属性仅适用于使用选择路径获得三维路径。
- 不管是三维路径还是二维路径，PathSketch 都可用。

对于创建肘形机械风管，分段放样是很有用的。通过设置两个放样参数（图 3-83）并草绘一个弧形路径可创建分段放样（图 3-84）。

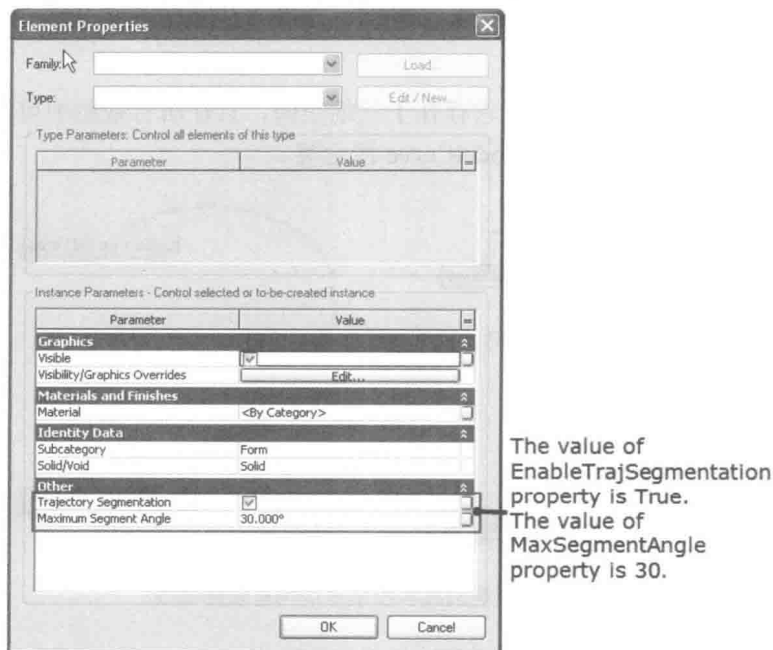


图 3-83 用户界面中对应的分段设置

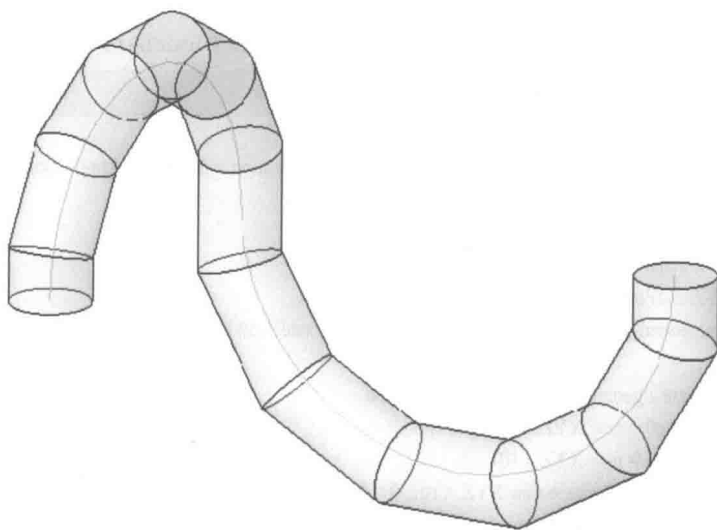


图 3-84 放样结果



注：以下信息适用于分段放样。

- 参数只影响路径上的弧。
- 放样所需最小分段数为 2。
- 通过清除轨迹分段复选框，可将分段放样更改为非分段放样，且 EnableTraj Segmentation 属性返回 “false”。
- 若 EnableTrajSegmentation 属性为 “false”，则最大分段角度值为默认值 360°。

### 3.8.3 模型曲线 (ModelCurve)

模型曲线表示项目中的模型线，它存在于三维空间，且在所有视图中可见。

图 3-85 和图 3-86 示出了四种 ModelCurve 派生类。



图 3-85 模型直线和模型弧线



图 3-86 模型椭圆线和模型非均匀有理样条线

#### 1. 创建模型曲线 (Creating a ModelCurve)

创建模型曲线的关键是创建 Geometry.Curve 和 Curve 所在的 SketchPlane。根据输入的 Geometry.Curve 类型，相应返回的 ModelCurve 可以向下转换为它的适当类型。

代码 3-68 阐释了如何新建模型曲线 (ModelLine 和 ModelArc)。

#### 代码 3-68：新建模型曲线

```
// get handle to application from document
Autodesk.Revit.ApplicationServices.Application application = document.Application;

// Create a geometry line in Revit application
XYZ startPoint = new XYZ (0, 0, 0);
XYZ endPoint = new XYZ (10, 10, 0);
Line geomLine = Line.CreateBound (startPoint, endPoint);

// Create a geometry arc in Revit application
XYZ end0 = new XYZ (1, 0, 0);
XYZ end1 = new XYZ (10, 10, 10);
XYZ pointOnCurve = new XYZ (10, 0, 0);
Arc geomArc = Arc.Create (end0, end1, pointOnCurve);

// Create a geometry plane in Revit application
```



```
XYZ origin = new XYZ (0, 0, 0);
XYZ normal = new XYZ (1, 1, 0);
Plane geomPlane = application.Create.NewPlane (normal, origin);

// Create a sketch plane in current document
SketchPlane sketch = SketchPlane.Create (document, geomPlane);

// Create a ModelLine element using the created geometry line and sketch plane
ModelLine line = document.Create.NewModelCurve (geomLine, sketch) as ModelLine;

// Create a ModelArc element using the created geometry arc and sketch plane
ModelArc arc = document.Cree.NewModelCurve (geomArc, sketch) as ModelArc;
```

## 2. 成员 (Members)

(1) 几何曲线 (GeometryCurve)。GeometryCurve 属性用于获取或设置模型曲线的几何曲线。除了 ModelHermiteSpline (模型厄米样条曲线) 以外, 可以从以下四种 ModelCurves 中得到不同的 Geometry.Curves:

- Line (直线)。
- Arc (弧线)。
- Ellipse (椭圆)。
- Nurbspline (非均匀有理样条线)。

代码 3-69 阐释了如何从模型曲线获得一个特定曲线。

### 代码 3-69: 从模型曲线获得特定曲线

```
//get the geometry modelCurve of the model modelCurve
Autodesk.Revit.DB.Curve geoCurve = modelCurve.GeometryCurve;

if (geoCurve is Autodesk.Revit.DB.Line)
{
    Line geoLine = geoCurve as Line;
}
```

GeometryCurve 属性的返回值是一个泛型 Geometry.Curve 对象, 因此, 转换对象类型必须使用 “As” 操作符。

注: 以下信息适用于几何曲线。

- 在 Revit 中无法创建厄米曲线, 但可以从其他软件如 AutoCAD 导入。Geometry.Curve 是唯一能代表厄米曲线的几何类。
- SetPlaneAndCurve() 方法和 Curve 及 SketchPlane 属性设置器适用于不同场合。
  - 若新曲线位于同一草图平面, 或新草图平面与原草图平面位于同一平面表面, 则使用 Curve 或 SketchPlane 属性设置器。
  - 如果新曲线位于不同的草图平面, 或新草图平面与原草图平面位于不同的平面表面, 则必须使用 SetPlaneAndCurve() 同时更改 Curve 值和 SketchPlane 值, 以避免内部数据不一致。



(2) 线样式 (Line Styles)。线样式是由 GraphicsStyle 类来表示。模型曲线的所有线样式都可以从 GetLineStyleIds() 方法返回一组 GraphicsStyle 图元的 ElementIds 集合来获取。

### 3.9 材料 (Material)

在 Revit 平台 API 中, 会存储材料数据并作为 Element 来管理。正如在 Revit 用户界面中一样, 材料可以有几种与之相关的“属性资源”(assets), 但只有热和结构 (在 Revit 用户界面中称为物理)“属性资源”可以由 API 指定。

某些材料的特性可由 Material 类本身属性来表示, 如 FillPattern (填充图案)、Color (颜色) 或 Render (渲染), 而其他特性可作为与材料相关的结构或热“属性资源”的属性。

在本节中, 将了解如何访问材料图元和如何管理文件中的 Material 对象。代码 3-77 演示了如何获取窗的材料。

#### 3.9.1 一般材料信息 (General Material Information)

如代码 3-70 所有 Material 对象可使用 Material 类过滤来检索。Material 对象也可从诸如 Document、Category、Element、Face 等获取, 这将在本节中的相关部分讨论。无论材料对象是从哪得到的, 都表示为 Material 类。

##### 1. 属性 (Properties)

材料会在某些方面与渲染外观、结构或其他主要的材料类别有关, 每个方面都会由 Material 属性类本身或其结构或热“属性资源”之一来表示。StructuralAsset 类表示与结构分析相关的材料属性。ThermalAsset 类表示与能量分析相关的材料属性。

代码 3-70: 获取材料属性

```
private void ReadMaterialProps (Document document, Material material)
{
    ElementId strucAssetId = material.StructuralAssetId;
    if (strucAssetId != ElementId.InvalidElementId)
    {
        PropertySetElement pse = document.GetElement (strucAssetId) as PropertySetElement;
        if (pse != null)
        {
            StructuralAsset asset = pse.GetStructuralAsset();

            // Check the material behavior and only read if Isotropic
            if (asset.Behavior == StructuralBehavior.Isotropic)
            {
                // Get the class of material
                StructuralAssetClass assetClass = asset.StructuralAssetClass; // Get other material properties

                // Get other material properties
                double poisson = asset.PoissonRatio.X;
```



```

double youngMod = asset.YoungModulus.X;
double thermCoeff = asset.ThermalExpansionCoefficient.X;
double unitweight = asset.Density;
double shearMod = asset.ShearModulus.X;
double dampingRatio = asset.DampingRatio;
if (assetClass == StructuralAssetClass.Metal)
{
    double dMinStress = asset.MinimumYieldStress;
}
elseif (assetClass == StructuralAssetClass.Concrete)
{
    double dConcComp = asset.ConcreteCompression;
}
}
}
}
}

```

## 2. 分类 (Classification)

与结构分析(即钢材、混凝土、木材)相关的材料分类,可从与材料关联的 `StructuralAsset` 的 `StructuralAssetClass` 属性获取。

注意: API 并不提供对 `Concrete` 材料的 `Concrete Type` 值的访问。

与能量分析(即固体、液体、气体)相关的材料分类,可从与材料关联的 `ThermalAsset` 的 `ThermalMaterialType` 属性获得。

## 3. 其他属性 (Other Properties)

材料对象属性标识特定的材料类型,包括颜色、填充图案,等等。

(1) 属性和参数 (Properties and Parameter)。某些 `Material` 属性只可作为 `Parameter` 使用。而另一些如 `Color`, 既可作为属性,也可用内建参数 `MATERIAL_PARAM_COLOR` 作为 `Parameter` 使用。

(2) 渲染信息 (Rendering Information)。分配到对象中的渲染数据的集合称为“属性资源”,它是只读的。从 `Application.Assets` 属性可以获得所有与外观相关的可用“属性资源”。通过 `Material.RenderAppearance` 属性可以访问材料的外观“属性资源”。

图 3-87 显示了“属性资源”浏览器对话框的外观库部分,以展示部分渲染“属性资源”在用户界面中是如何显示的。

SDK 中附带的 `Materials` 示例应用程序演示了如何设置对话框中所选材料的 `Render Appearance` 属性。对话框中显示了 `Application.Assets` 中所有 `Asset` 对象。

(3) 填充图案 (FillPattern)。使用 `FilteredElementCollector` 过滤 `FillPatternElement` 类可获得文件中所有填充图案。`FillPatternElement` 是一个包含 `FillPattern` 的图元,而 `FillPattern` 类提供对图案名称和组成图案的 `FillGrids` 设置的访问。

有两种填充图案: `Drafting` 和 `Model`。在用户界面中,只能对 `Material.CutPattern` 设置 `Drafting` 填充图案。填充图案类型通过 `FillPattern.Target` 属性公开,代码 3-71 演示了如何更改材料的 `FillPattern`。



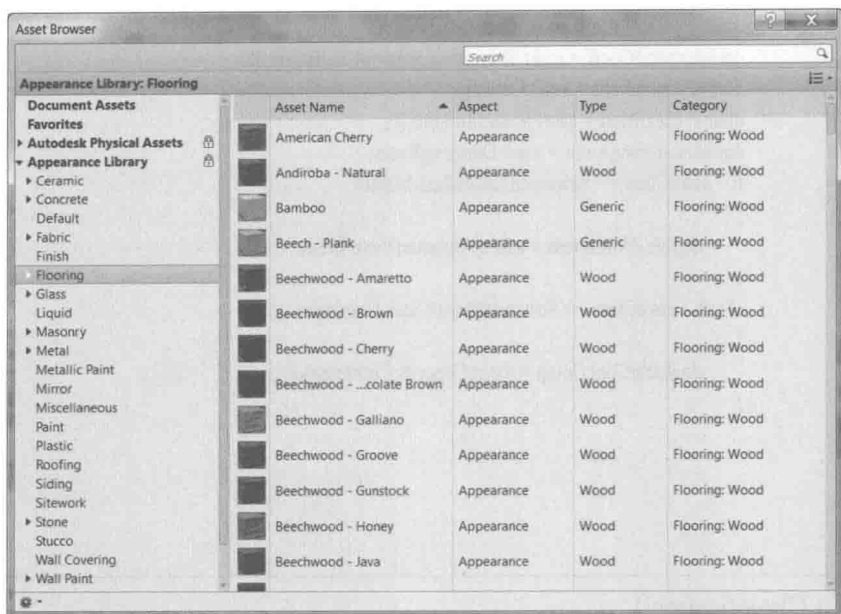


图 3-87 外观库

**代码 3-71: 设置填充图案**

```
public void SetFillPattern (Document document, Material material)
{
    FilteredElementCollector collector = new FilteredElementCollector (document);
    ICollection<FillPatternElement> fillPatternElements =

    collector.OfClass (typeof (FillPatternElement)).Cast<FillPatternElement>().ToList();
    foreach (FillPatternElement fillPattern in fillPatternElements)
    {
        // always set successfully
        material.CutPattern = fillPattern;
        material.SurfacePattern = fillPattern;
    }
}
```

**3.9.2 材料管理 (Material Management)**

可以使用过滤器来检索文件中的所有材料。Document 中的每个 Material 对象都有唯一名称标识。

代码 3-72 阐释了如何使用材料名称来获取材料。

**代码 3-72: 由名称获取材料**

```
FilteredElementCollector elementCollector = new FilteredElementCollector (document);
elementCollector.WherePasses (new ElementClassFilter (typeof (Material)));
IList<Element> materials = elementCollector.ToElements();

Material floorMaterial = null;
```



```
string floorMaterialName = "Default Floor";

foreach (Element materialElement in materials)
{
    Material material = materialElement as Material;
    if (floorMaterialName == material.Name)
    {
        floorMaterial = material;
        break;
    }
}
if (null != floorMaterial)
{
    TaskDialog.Show ("Revit", "Material found.");
}
```

注：要运行代码 3-72，须确保文件中已有该材料名称。当前文件的所有材料名称都位于管理选项卡下（项目设置面板 ► 材料）。

#### 1. 创建材料 (Creating Materials)

API 中有两种方法可创建新的 Material 对象：

- 复制现有 Material，见代码 3-73。
- 添加新 Material。

当使用 Duplicate() 方法时，返回的 Material 对象与被复制的 Material 为同一类型。

#### 代码 3-73：材料复制

```
private bool DuplicateMaterial (Material material)
{
    bool duplicated = false;
    //try to duplicate a new instance of Material class using duplicate method
    //make sure the name of new material is unique in MaterialSet
    string newName = "new" + material.Name;
    Material myMaterial = material.Duplicate (newName);
    if (null == myMaterial)
    {
        TaskDialog.Show ("Revit", "Failed to duplicate a material!");
    }
    else
    {
        duplicated = true;
    }

    return duplicated;
}
```

使用静态方法 Material.Create() 来直接添加新材料，见代码 3-74。不管它是如何应用的，必须为材料指定唯一名称，以及属于该材料的任一“属性资源”。唯一名称是 Material 对象的关键。



### 代码 3-74: 添加新材料

```
//Create the material
ElementId materialId = Material.Create ( document, "My Material");
Material material = document.GetElement (materialId) as Material;

//Create a new property set that can be used by this material
StructuralAsset strucAsset = new StructuralAsset ( "My Property Set", StructuralAssetClass.Concrete);
strucAsset.Behavior = StructuralBehavior.Isotropic;
strucAsset.Density = 232.0;

//Assign the property set to the material.
PropertySetElement pse = PropertySetElement.Create ( document, strucAsset);
material.SetMaterialAspectByPropertySet (MaterialAspect.Structural, pse.Id);
```

## 2. 删除材料 ( Deleting Materials )

要删除材料, 请使用 `Document.Delete()` 方法。 `Document.Delete()` 是个泛型方法。有关详细信息, 请参阅 2.5 节编辑图元。

### 3.9.3 图元材料 ( Element Material )

一个图元可以包含多个图元和构件 (如图 3-88 中的栏杆构件)。例如, `FamilyInstance` 可含有 `SubComponents`, `Wall` 可有包含几个 `CompoundStructure Layers` 的 `CompoundStructure` (有关子构件的详细信息请参阅 3.2 节族实例, 并参阅 3.1 节墙、楼板、屋顶和洞口以获得复合结构的更多信息)。

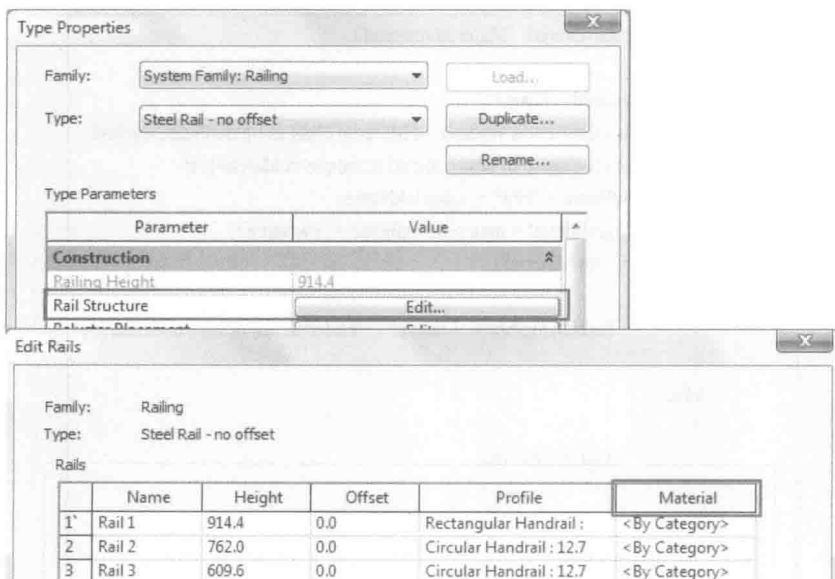


图 3-88 栏杆构件的属性

Revit 平台 API 中, 获取图元材料使用下列准则:

- 若图元包含其他图元, 则分别获取材料。



- 若图元包含构件，则由参数或特定方式（请参阅 3.1 节墙、楼板、屋顶和洞口的材料章节）获取每个构件的材料。
- 若构件的材料返回 null，则从相应的 Element.Category 子 Category 获取材料。

#### 1. 参数中的材料 (Material in a Parameter)

如果 Element 对象有一个 ParameterType 为 ParameterType.Material 的 Parameter，则可从该 Parameter 获得图元的材料。例如，结构柱 FamilySymbol (其 Category 为 BuiltInCategory.OST\_StructuralColumns 的族实例) 具有 Structural Material 参数，可使用 ElementId 获取其 Material。代码 3-75 演示了如何获取具有一个构件的结构柱材料。

**代码 3-75: 由参数获取图元材料**

```
public void GetMaterial (Document document, FamilyInstance familyInstance)
{
    foreach (Parameter parameter in familyInstance.Parameters)
    {
        Definition definition = parameter.Definition;
        // material is stored as element id
        if (parameter.StorageType == StorageType.ElementId)
        {
            if (definition.ParameterGroup == BuiltInParameterGroup.PG_MATERIALS &&
                definition.ParameterType == ParameterType.Material)
            {
                Autodesk.Revit.DB.Material material = null;
                Autodesk.Revit.DB.ElementId materialId = parameter.AsElementId();
                if (-1 == materialId.IntegerValue)
                {
                    //Invalid ElementId, assume the material is "By Category"
                    if (null != familyInstance.Category)
                    {
                        material = familyInstance.Category.Material;
                    }
                }
                else
                {
                    material = document.GetElement (materialId) as Material;
                }

                TaskDialog.Show ("Revit", "Element material: " + material.Name);
                break;
            }
        }
    }
}
```

注：在用户界面中，如果材料属性设置为“按类别”，则此材料的 ElementId 为 ElementId.InvalidElementId，不能用于检索 Material 对象，如代码 3-75 所示。请尝试由 Category 检索 Material，如后所述。



有些包含于其他复合参数中的材料属性,无法从 API 访问。作为例子,在图 3-88 中,对于系统族——Railing,其 Rail Structure 参数的 StorageType 为 StorageType.None。因此,这种情况下无法得到材料信息。

## 2. 材料和族实例 (Material and FamilyInstance)

对于梁、柱和基础族实例,使用 StructuralMaterialId 属性,是获取其材料的另一方式,见代码 3-76。此属性返回一个 ElementId,来标识定义实例结构分析属性的材料。

代码 3-76: 由族实例获取图元材料

```
public Material GetFamilyInstanceMaterial (Document document, FamilyInstance beam)
{
    Material material = document.GetElement (beam.StructuralMaterialId) as Material;

    return material;
}
```

## 3. 材料和类别 (Material and Category)

只有模型图元才有“材料”,从 Revit 管理选项卡上,单击设置 ➤ 对象样式来显示对象样式对话框,其类别列于模型对象选项卡中的图元具有材料信息 (图 3-89)。

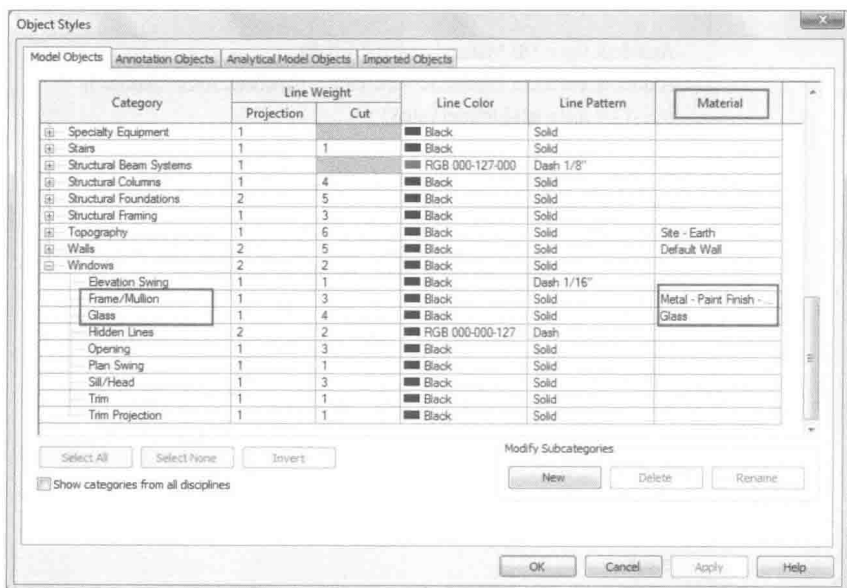


图 3-89 类别材料

只有 Model 图元才可以指定 Material 属性。因此,若查询其类别不是 Model 的其他图元(如 Annotations 或 Imported)的 Material,其结果总是 null。有关 Element 和 Category 分类的更多细节,请参阅 1.5 节图元概要。

如果某图元包含多个构件,则其 Category.Subcategories 中的某些是 Material 对应于构件。在图 3-89 对象样式对话框中,“Windows”类别和“Frame/Mullion”及“Glass”子类



别映射到 Windows 图元的构件。图 3-90 中, 窗符号 Glass Pane Material 参数似乎是得到窗玻璃材料的唯一方式。然而, 其值是“按类别”, 相应的 Parameter 返回 ElementId.InvalidElementId。

在这种情况下, 该窗格的 Material 其实并不为 null, 而是取决于窗类别 OST\_Windows 的子类别 OST\_WindowsFrameMullionProjection 的 Material 属性。如果它仍然返回 null, 则窗格的材料取决于父类别 OST\_Windows。有关详细信息, 请参阅代码 3-77: 获取窗的材料演示。

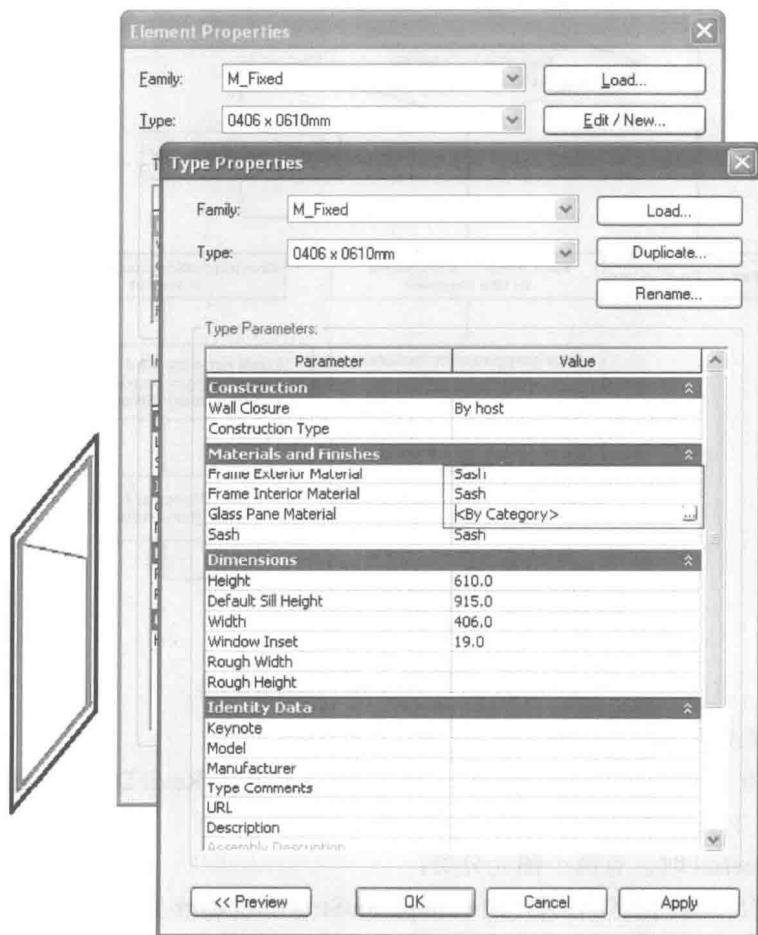


图 3-90 窗的材料

#### 4. 复合结构层材料 (Compound StructureLayer Material)

从 HostObjAttributes 可以获取复合结构层对象。有关详细信息, 请参阅 3.1 节墙、楼板、天花板、屋顶和洞口。

#### 5. 检索图元材料 (Retrieve Element Materials)

图 3-91 显示了检索图元材料的工作流程。



图 3-91 获取图元材料的工作流程

此工作流程说明了以下过程:

- 如何获取属于图元的 Material 对象 (非 Autodesk.Revit.DB. Structure. Structural Material Type 枚举类型)。
- 检索 Material 时, 有两个图元分类:
  - 含有复合结构的主体对象: 由 CompoundStructureLayer 类的 MaterialId 属性获取材料对象。
  - 其他: 由 Parameters 获取材料。
- 当获取的材料对象为 null 或 ElementId.InvalidElementId 值为“无效 ElementId”时, 请尝试由相应的类别获取 Material。要注意的是, 族实例与其族符号通常具有相同的类别。
- 对 Element 对象越了解, 获取材料也就越容易。例如:
  - 如果知道这个图元是梁, 那么就可获取其 Structural Material 实例参数。
  - 若知道这个图元是窗, 则可将其归为 FamilyInstance 并获取其 FamilySymbol。
- 还可以通过获取诸如 Frame Exterior Material 或 Frame Interior Material 的



Parameters 来获取 Material 对象。如果所获取的是 null，请尝试从族符号类别获取材料对象。

- 并非所有图元材料都可由 API 获取。

#### 6. 演练：获取窗的材料 (Walkthrough: Get Window Materials)

代码 3-77 说明了如何获取窗的材料。

#### 代码 3-77：获取窗的材料演示

```
public void GetMaterial (Document document, FamilyInstance window)
{
    Materials materials = document.Settings.Materials;
    FamilySymbol windowSymbol = window.Symbol;
    Category category = windowSymbol.Category;
    Autodesk.Revit.DB.Material frameExteriorMaterial = null;
    Autodesk.Revit.DB.Material frameInteriorMaterial = null;
    Autodesk.Revit.DB.Material sashMaterial = null;
    // Check the paramters first
    foreach (Parameter parameter in windowSymbol.Parameters)
    {
        switch (parameter.Definition.Name)
        {
            case "Frame Exterior Material":
                frameExteriorMaterial = materials.get_Item (parameter.AsElementId());
                break;
            case "Frame Interior Material":
                frameInteriorMaterial = materials.get_Item (parameter.AsElementId());
                break;
            case "Sash":
                sashMaterial = materials.get_Item (parameter.AsElementId());
                break;
            default:
                break;
        }
    }
    // Try category if the material is set by category
    if (null == frameExteriorMaterial)
        frameExteriorMaterial = category.Material;
    if (null == frameInteriorMaterial)
        frameInteriorMaterial = category.Material;
    if (null == sashMaterial)
        sashMaterial = category.Material;
    // Show the result because the category may have a null Material,
    // the Material objects need to be checked.
    string materialsInfo = "Frame Exterior Material: " + (null != frameExteriorMaterial ? frameExteriorMaterial.Name :
    "null") + "\n";
    materialsInfo += "Frame Interior Material: " + (null != frameInteriorMaterial ? frameInteriorMaterial.Name : "null")
    + "\n";
    materialsInfo += "Sash: " + (null != sashMaterial ? sashMaterial.Name : "null") + "\n";
    TaskDialog.Show ("Revit", materialsInfo);
}
```





### 3.9.4 材料数量 (Material Quantities)

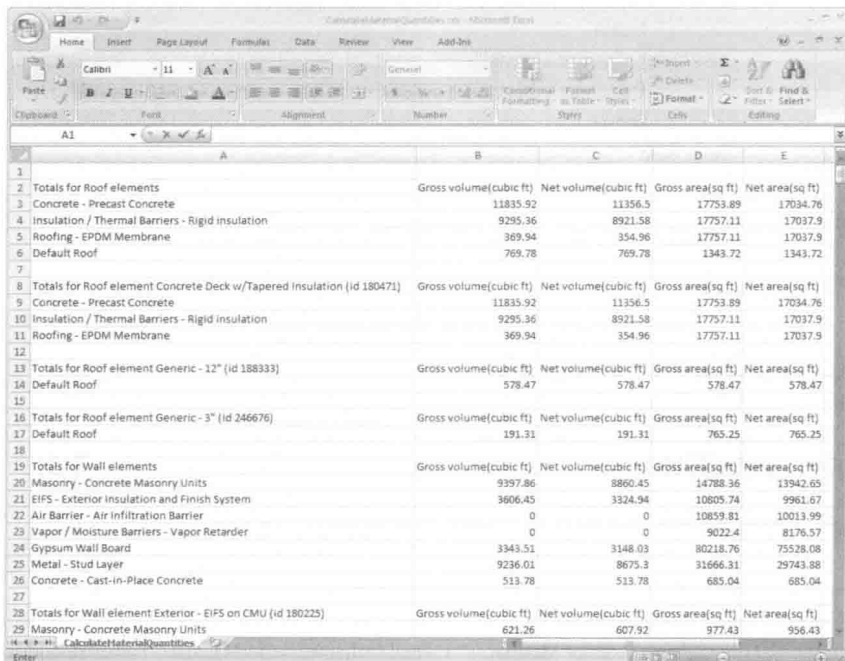
获取 Revit 材料明细表中材料体积和面积的直接方法:

- `Element.Materials`: 得到图元内部的材料清单。
- `Element.GetMaterialVolume()`: 得到图元中特定材料的体积。
- `Element.GetMaterialArea()`: 得到图元中特定材料的面积。

该方法应用于 `Category.HasMaterialQuantities` 属性为 `true` 的图元类别。实际上,这仅限于使用复合结构的图元,如墙、屋顶、楼板、天花板,其他一些基本的三维图元,如楼梯,再加上可对族几何体指定材料的三维族,如门、窗、柱、MEP 设备和固定装置,以及一般模型族。请注意,在这些类别内,对如何提取材料的数量有进一步的限制。例如,幕墙和幕墙屋顶自身不报告任何材料数量;用于这些结构的材料可以从组成幕墙系统的单个嵌板图元中提取。

要注意,在某些情况下,Revit 计算的体积和面积可能是近似的。例如,对于墙内的单个层,模型中可见的体积与材料提取明细表中显示的体积两者之间可能会有些微小差异。这些差异往往在使用“墙饰条”工具将墙饰条或分隔缝添加到墙上时或者存在某些连接的条件上发生。

SDK 示例“MaterialQuantities”结合材料数量提取工具和暂时禁止切割图元(洞口、窗、门)来提取材料毛数量和净数量,结果见图 3-92。



	Gross volume(cubic ft)	Net volume(cubic ft)	Gross area(sq ft)	Net area(sq ft)
<b>Totals for Roof elements</b>				
Concrete - Precast Concrete	11835.92	11356.5	17753.89	17034.76
Insulation / Thermal Barriers - Rigid Insulation	9295.36	8921.58	17757.11	17037.9
Roofing - EPDM Membrane	369.94	354.96	17757.11	17037.9
Default Roof	769.78	769.78	1343.72	1343.72
<b>Totals for Roof element Concrete Deck w/Tapered Insulation (id 180471)</b>				
Concrete - Precast Concrete	11835.92	11356.5	17753.89	17034.76
Insulation / Thermal Barriers - Rigid Insulation	9295.36	8921.58	17757.11	17037.9
Roofing - EPDM Membrane	369.94	354.96	17757.11	17037.9
<b>Totals for Roof element Generic - 12" (id 188333)</b>				
Default Roof	578.47	578.47	578.47	578.47
<b>Totals for Roof element Generic - 3" (id 246676)</b>				
Default Roof	191.31	191.31	765.25	765.25
<b>Totals for Wall elements</b>				
Masonry - Concrete Masonry Units	9397.86	8860.45	14788.36	13942.65
EIFS - Exterior Insulation and Finish System	3606.45	3324.94	10805.74	9961.67
Air Barrier - Air Infiltration Barrier	0	0	10859.81	10013.99
Vapor / Moisture Barriers - Vapor Retarder	0	0	9022.4	8176.57
Gypsum Wall Board	3343.51	3148.03	80218.76	75528.08
Metal - Stud Layer	9236.01	8675.3	31666.31	29743.88
Concrete - Cast-in-Place Concrete	513.78	513.78	685.04	685.04
<b>Totals for Wall element Exterior - EIFS on CMU (id 180225)</b>				
Masonry - Concrete Masonry Units	621.26	607.92	977.43	956.43

图 3-92 提取材料毛数量和净数量结果

### 3.9.5 涂装图元表面 (Painting the Face of an Element)

通过 Revit API 可使用 Paint 工具功能,见代码 3-78。图元表面如墙面、楼面和屋面可涂覆材料以改变其外观。但它不会更改图元的结构。



代码 3-78: 涂装墙面

```
// Paint any unpainted faces of a given wall
public void PaintWallFaces (Wall wall, ElementId matId)
{
    Document doc = wall.Document;
    GeometryElement geometryElement = wall.get_Geometry (new Options());
    foreach (GeometryObject geometryObject in geometryElement)
    {
        if (geometryObject is Solid)
        {
            Solid solid = geometryObject as Solid;
            foreach (Face face in solid.Faces)
            {
                if (doc.IsPainted (wall.Id, face) == false)
                {
                    doc.Paint (wall.Id, face, matId);
                }
            }
        }
    }
}
```

涂装图元相关的方法是 Document 类的一部分。Document.Paint() 将材料应用到指定图元面。Document.RemovePaint() 将删除已应用的材料。此外, IsPainted() 和 GetPaintedMaterial() 方法返回有关图元表面的信息。

## 3.10 楼梯和栏杆扶手 (Stairs and Railings)

### 3.10.1 楼梯 (Stairs)

在 Autodesk.Revit.DB.Architecture 命名空间中 Revit API 的类允许访问楼梯和相关构件, 如楼梯平台和梯段。可以使用 Revit API 创建或修改楼梯。Stairs 类表示楼梯“按构件”创建。按草图创建的楼梯图元, 在 API 中无法作为 Stair 对象访问。静态方法 Stairs.IsByComponent() 可用于确定 ElementId 所表示的楼梯是否按构件创建。

#### 1. 楼梯编辑作用域 (StairsEditScope)

正如 Revit 文件中其他类型图元一样, 需要一个用于编辑楼梯和楼梯构件的事物 (Transaction)。然而, 若要创建新的构件, 如梯段和楼梯平台, 或创建新楼梯本身, 还需要使用一个可维持楼梯编辑会话的 Autodesk.Revit.DB.StairsEditScope 对象。

StairsEditScope 就像个事物组。StairsEditScope 启动后, 才可启动事务来编辑楼梯。在 StairsEditScope 内创建的独特事务不会出现在“撤销”菜单中。在编辑模式期间提交的所有事务将合并为一个事务, 该事务带有传递到 StairsEditScope 构造函数的事务名称。

StairsEditScope 有两种启动方法。第一种方法, 需要一个现有 Stairs 对象的 ElementId,



以对其启动编辑会话；第二种方法，需要底部及顶部标高 `ElementIds`，以默认楼梯类型于指定标高新建一个“空”楼梯图元，然后对其启动楼梯编辑模式。

在梯段和平台添加到 `Stairs` 并完成编辑之后，调用 `StairsEditScope.Commit()` 结束楼梯编辑会话。

## 2. 添加梯段 (Adding Runs)

`StairsRun` 类为楼梯对象创建新梯段有三个静态方法：

- `CreateSketchedRun`；按提供的一组边界曲线和踢面曲线创建草图梯段。
- `CreateStraightRun`；创建直梯段。
- `CreateSpiralRun`；按提供的圆心、起始角和夹角，创建螺旋梯段。

## 3. 添加楼梯平台 (Adding Landings)

可以在两个梯段之间自动添加或草绘添加楼梯平台。静态方法 `StairsLanding.CanCreateAutomaticLanding()` 用来检查两个楼梯梯段是否满足自动创建楼梯平台的限制条件。静态方法 `StairsLanding.CreateAutomaticLanding()` 将返回两个楼梯梯段之间所创建的所有平台的 ID。

静态方法 `StairsLanding.CreateSketchedLanding()`，根据所提供的闭合平台边界曲线，在两个梯段之间创建一个自定义楼梯平台。`CreateSketchedLanding` 方法的输入之一是个双精度型底部标高值。对标高有下列限制：

- 底部标高相对于楼梯底部标高。
- 底部标高会按踢面高度的倍数四舍五入自动凑整。
- 底部标高应不小于半个踢面高度。

## 4. 示例 (Example)

代码 3-79 创建了一个新楼梯对象、两个梯段（一个草绘梯段和一个直梯段），并在梯段之间设一个楼梯平台。

代码 3-79：创建楼梯、梯段和楼梯平台

```
private ElementId CreateStairs (Document document, Level levelBottom, Level levelTop)
{
    StairsEditScope newStairsScope = new StairsEditScope (document, "New Stairs");
    ElementId newStairsId = newStairsScope.Start (levelBottom.Id, levelTop.Id);

    Transaction stairsTrans = new Transaction (document, "Add Runs and Landings to Stairs");
    stairsTrans.Start();

    // Create a sketched run for the stairs
    IList<Curve> bdryCurves = new List<Curve>();
    IList<Curve> riserCurves = new List<Curve>();
    IList<Curve> pathCurves = new List<Curve>();
    XYZ pnt1 = new XYZ (0, 0, 0);
    XYZ pnt2 = new XYZ (15, 0, 0);
    XYZ pnt3 = new XYZ (0, 10, 0);
    XYZ pnt4 = new XYZ (15, 10, 0);

    // boundaries
```



```

bdryCurves.Add (Line.get_Bound (pnt1, pnt2));
bdryCurves.Add (Line.get_Bound (pnt3, pnt4));

// riser curves
const int riserNum = 20;
for (int ii = 0; ii <= riserNum; ii++)
{
    XYZ end0 = (pnt1 + pnt2) * ii / (double) riserNum;
    XYZ end1 = (pnt3 + pnt4) * ii / (double) riserNum;
    XYZ end2 = new XYZ (end1.X, 10, 0);
    riserCurves.Add (Line.get_Bound (end0, end2));
}

//stairs path curves
XYZ pathEnd0 = (pnt1 + pnt3) / 2.0;
XYZ pathEnd1 = (pnt2 + pnt4) / 2.0;
pathCurves.Add (Line.get_Bound (pathEnd0, pathEnd1));

StairsRun newRun1 = StairsRun.CreateSketchedRun (document, newStairsId, levelBottom.Elevation, bdryCurves,
riserCurves, pathCurves);

// Add a straight run
Line locationLine = Line.get_Bound (new XYZ (20, -5, newRun1.TopElevation), new XYZ (35, -5,
newRun1.TopElevation));
StairsRun newRun2 = StairsRun.CreateStraightRun (document, newStairsId, locationLine, StairsRunJustification.Center);
newRun2.ActualRunWidth = 10;

// Add a landing between the runs
CurveLoop landingLoop = new CurveLoop();
XYZ p1 = new XYZ (15, 10, 0);
XYZ p2 = new XYZ (20, 10, 0);
XYZ p3 = new XYZ (20, -10, 0);
XYZ p4 = new XYZ (15, -10, 0);
Line curve_1 = Line.get_Bound (p1, p2);
Line curve_2 = Line.get_Bound (p2, p3);
Line curve_3 = Line.get_Bound (p3, p4);
Line curve_4 = Line.get_Bound (p4, p1);

landingLoop.Append (curve_1);
landingLoop.Append (curve_2);
landingLoop.Append (curve_3);
landingLoop.Append (curve_4);
StairsLanding newLanding = StairsLanding.CreateSketchedLanding (document, newStairsId, landingLoop,
newRun1.TopElevation);

stairsTrans.Commit();
newStairsScope.Commit();

return newStairsId;
}

```



由代码 3-79 产生的楼梯见图 3-93。

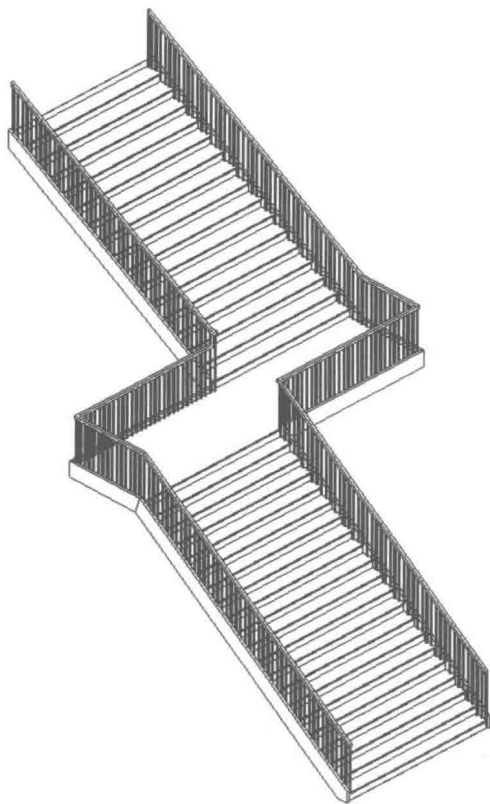


图 3-93 API 代码产生的楼梯

### 5. 添加栏杆扶手 (Adding Railings)

在使用 `StairsEditScope.Start (ElementId, ElementId)` 方法创建新楼梯时, 会带有与其关联的默认栏杆扶手。不过, 对于尚无栏杆扶手的楼梯, 可用 `Railing.Create()` 方法在楼梯图元的每一侧, 创建具有指定类型的新栏杆扶手, 见代码 3-80。不同于梯段和平台创建需要使用 `StairsEditScope`, 在打开的楼梯编辑会话内是无法执行栏杆创建的。

#### 代码 3-80: 创建栏杆

```
private void CreateRailing (Document document, Stairs stairs)
{
    Transaction trans = new Transaction (document, "Create Railings");
    trans.Start();

    // Delete existing railings
    ICollection<ElementId> railingIds = stairs.GetAssociatedRailings();
    foreach (ElementId railingId in railingIds)
    {
        document.Delete (railingId);
    }
    // Find RailingType
```



```
FilteredElementCollector collector = new FilteredElementCollector (document);
ICollection<ElementId> RailingTypeIds = collector.OfClass (typeof (RailingType)).ToElementIdsElementId();
Railing.Create (document, stairs.Id, RailingTypeIds.First(), RailingPlacementPosition.Treads);
trans.Commit();
}
```

### 3.10.2 栏杆扶手 (Railings)

Autodesk. Revit. DB. Architecture. Railing 类表示 Autodesk Revit 项目中的栏杆扶手图元。虽然栏杆扶手与楼梯相关，但也可以有其他主体，如楼板，或放置在空地上。栏杆扶手可以是连续的，也可以是不连续的。对不连续的栏杆扶手，仅提供有限层级的访问。

与楼梯关联的栏杆扶手可以用 `GetAssociatedRailings` 方法从 `Stairs` 类来检索。只有少数几个特定于栏杆扶手的属性和方法，如 `TopRail` 属性返回顶部扶栏的 `ElementId`，`Flipped` 表示栏杆扶手是否被翻转了。`Railing.Flip()` 方法翻转栏杆扶手，而 `RemoveHost` 方法会删除栏杆扶手与主体之间的关联。

代码 3-81 检索了与 `Stairs` 对象相关的所有栏杆扶手，并翻转那些由系统默认生成的每个栏杆扶手。

**代码 3-81: 栏杆操作**

```
private void FlipDefaultRailings (Stairs stairs)
{
    ICollection<ElementId> railingIds = stairs.GetAssociatedRailings();
    Transaction trans = new Transaction (stairs.Document, "Flip Railings");
    trans.Start();
    foreach (ElementId railingId in railingIds)
    {
        Railing railing = stairs.Document.GetElement (railingId) as Railing;
        if (railing.IsDefault == true)
        {
            railing.Flip();
        }
    }
    trans.Commit();
}
```

`Railing` 类有一个 `Create` 方法，在楼梯图元的每一侧，以指定类型自动创建新栏杆扶手，

3.10.1 节楼梯中演示了栏杆扶手创建。

`RailingType` 类表示用于生成栏杆扶手的栏杆扶手类型。它包含了很多栏杆扶手属性，如高度、侧向偏移、主扶手和辅助扶手类型，以及顶部扶栏，见代码 3-82。

**代码 3-82: 栏杆类型**

```
private void GetRailingType (Stairs stairs)
{
    ICollection<ElementId> railingIds = stairs.GetAssociatedRailings();
    foreach (ElementId railingId in railingIds)
    {
        Railing railing = stairs.Document.GetElement (railingId) as Railing;
```



```
RailingType railingType = stairs.Document.GetElement (railing.GetTypeId()) as RailingType;

// Format railing type info for display
string info = "Railing Type:  " + railingType.Name;
info += "\nPrimary Handrail Height:  " + railingType.PrimaryHandrailHeight;
info += "\nTop Rail Height:  " + railingType.TopRailHeight;

TaskDialog.Show ("Revit", info);
}
}
```

### 3.10.3 楼梯注释 (Stairs Annotations)

StairsPath 类可用于注释楼梯的坡度方向和行走线。静态方法 StairsPath.Create() 在指定的楼梯可见平面视图中, 以指定路径类型新建该楼梯的楼梯路径。

StairsPath 类的属性与在 Revit 用户界面中编辑楼梯路径的属性窗中看到的属性相同, 如图 3-94 中用于设置“向上”和“向下”文字的属性, 或究竟是否要显示文字的属性。此外, “向上”和“向下”文字的偏移可指定为楼梯路径至楼梯中心线的偏移量。

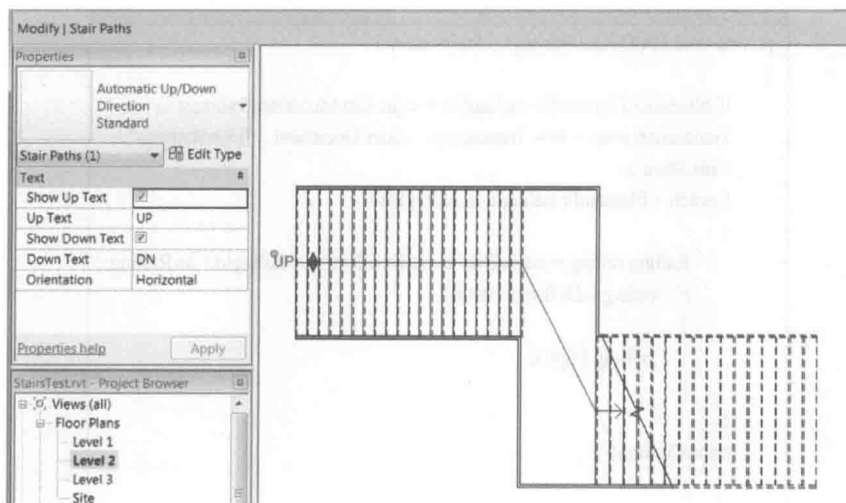


图 3-94 StairsPath 类的属性

代码 3-83 查找了项目中的楼梯路径类型和楼层平面图, 并用它们为给定楼梯创建新的楼梯路径。

#### 代码 3-83: 创建楼梯路径

```
private void CreateStairsPath (Document document, Stairs stairs)
{
    Transaction transNewPath = new Transaction (document, "New Stairs Path");
    transNewPath.Start();

    // Find StairsPathType
    FilteredElementCollector collector = new FilteredElementCollector (document);
```



```

ICollection<ElementId> stairsPathIds = collector.OfClass (typeof (StairsPathType)) .ToElementIdsElementId();

// Find a FloorPlan
ElementId planViewId = ElementId.InvalidElementId;
FilteredElementCollector viewCollector = new FilteredElementCollector (document);
ICollection<ElementId> viewIds = viewCollector.OfClass (typeof (View)) .ToElementIdsElementId();
foreach (ElementId viewId in viewIds)
{
    View view = document.GetElement (viewId) as View;
    if (view.ViewType == ViewType.FloorPlan)
    {
        planViewId = view.Id;
        break;
    }
}

LinkElementId stairsLinkId = new LinkElementId (stairs.Id);
StairsPath.Create (stairs.Document, stairsLinkId, stairsPathIds.First(), planViewId);
transNewPath.Commit();
}

```

StairsPath 具有 StairsPathType。楼梯路径类型可从两个预定义系统族得到：Automatic Up/Down Direction 和 Fixed Up Direction。这两种可用属性都可以作为 StairsPathType 类中的属性，如图 3-95 中的 FullStepArrow 和 DistanceToCutMark。

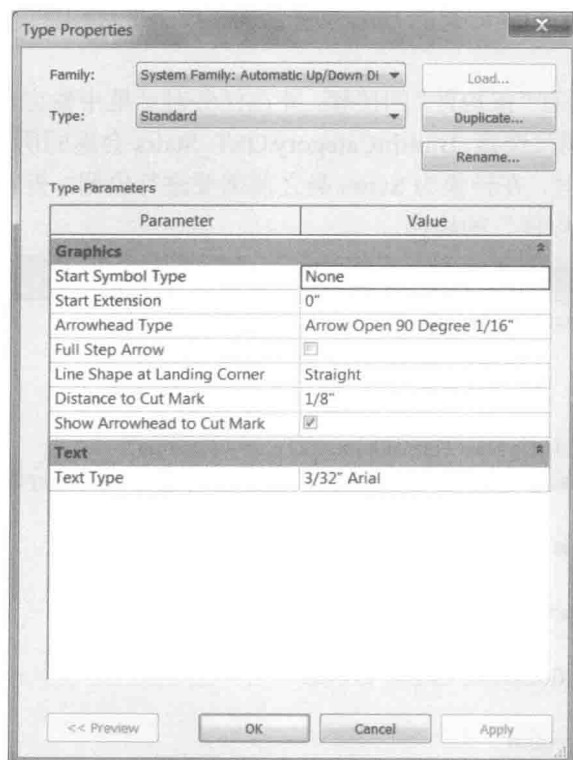


图 3-95 StairsPathType 类的属性





CutMarkType 类表示 Revit 用户界面中的剪切标记类型, 它包含的属性和在用户界面中编辑剪切标记类型时显示的属性相同, 如 CutLineAngle 和 CutLineExtension。它与 StairsType 对象相关联, 可以用内建参数 STAIRSTYPE\_CUTMARK\_TYPE 来检索, 见代码 3-84。

代码 3-84: 获取楼梯剪切标记类型 (CutMarkType)

```
private CutMarkType GetCutMark (Stairs stairs)
{
    CutMarkType cutMarkType = null;
    StairsType stairsType = stairs.Document.GetElement (stairs.GetTypeId()) as StairsType;
    Parameter paramCutMark = stairsType.get_Parameter (BuiltInParameter.STAIRSTYPE_CUTMARK_TYPE);
    if (paramCutMark.StorageType == StorageType.ElementId) // should be an element id
    {
        ElementId cutMarkId = paramCutMark.AsElementId();
        cutMarkType = stairs.Document.GetElement (cutMarkId) as CutMarkType;
    }

    return cutMarkType;
}
```

### 3.10.4 楼梯构件 (Stairs Components)

Stairs 类表示 Revit 中的楼梯图元并包含以下属性信息: 踏板、踢面、层数, 以及楼梯高度、底部和顶部标高。Stairs 类的方法可用于获取楼梯平台构件、楼梯梯段构件和楼梯的支撑。

代码 3-85 查找了所有“按构件”的楼梯, 并在任务对话框中输出每个楼梯的一些信息。请注意, 此示例使用类别筛选器, BuiltInCategory.OST\_Stairs 会返回所有楼梯的 ElementIds。因此, 当从文件中检索时, 在转换为 Stairs 类之前需要进行检测, 查看每个 ElementId 所表示的楼梯是否属于“按构件”的楼梯。

代码 3-85: 获取楼梯信息

```
private Stairs GetStairInfo (Document document)
{
    Stairs stairs = null;

    FilteredElementCollector collector = new FilteredElementCollector (document);
    ICollection<ElementId> stairsIds = collector.WhereElementIsNotElementType().OfCategory (BuiltInCategory.OST_Stairs).
    ToElementIdsElementId();
    foreach (ElementId stairId in stairsIds)
    {
        if (Stairs.IsByComponent (document, stairId) == true)
        {
            stairs = document.GetElement (stairId) as Stairs;

            // Format the information
            String info = "\nNumber of stories: " + stairs.NumberOfStories;
            info += "\nHeight of stairs: " + stairs.Height;
        }
    }
}
```



```

        info += "\nNumber of treads:  " + stairs.ActualTreadsNumber;
        info += "\nTread depth:  " + stairs.ActualTreadDepth;

        // Show the information to the user.
        TaskDialog.Show ("Revit", info);
    }
}

return stairs;
}

```

**StairsType** 类表示 **Stairs** 图元的类型。它包含有关 **Stairs** 的信息，如楼梯对象中所有梯段和平台的类型，楼梯左侧、右侧和中间支撑的类型和偏移，以及与楼梯生成相关的众多其他属性，如楼梯图元每个踢面的最大高度。代码 3-86 示例了如何获取楼梯的楼梯图元类型并在任务对话框中显示一些有关信息。

#### 代码 3-86：获取楼梯图元类型信息

```

private void GetStairsType (Stairs stairs)
{
    StairsType stairsType = stairs.Document.GetElement (stairs.GetTypeId()) as StairsType;

    // Format stairs type info for display
    string info = "Stairs Type:  " + stairsType.Name;
    info += "\nLeft Lateral Offset:  " + stairsType.LeftLateralOffset;
    info += "\nRight Lateral Offset:  " + stairsType.RightLateralOffset;
    info += "\nMax Riser Height:  " + stairsType.MaxRiserHeight;
    info += "\nMin Run Width:  " + stairsType.MinRunWidth;

    TaskDialog.Show ("Revit", info);
}

```

#### 1. 梯段 (Runs)

“按构件”楼梯由梯段、平台和支撑组成。每项构件都可从 **Stairs** 类检索。Revit API 中，梯段由 **StairsRun** 类来表示。代码 3-87 示例了如何获取楼梯对象的每个梯段，并确保它以踢面起始、以踢面结束。

#### 代码 3-87：使用 StairsRun

```

private void AddStartandEndRisers (Stairs stairs)
{
    ICollection<ElementId> runIds = stairs.GetStairsRuns();

    foreach (ElementId runId in runIds)
    {
        StairsRun run = stairs.Document.GetElement (runId) as StairsRun;
        if (null != run)
        {
            run.BeginsWithRiser = true;
        }
    }
}

```



```
run.EndsWithRiser = true;
    }
}
```

StairsRun 类支持对梯段属性的访问, 如 StairsRunStyle (直梯段、螺旋梯段等)、底部标高、顶部标高以及踢面相关属性。StairsRun 类中还含有用于访问以梯段为主体的支撑的方法, 包括全部或梯段左、右某一侧边界支撑。GetStairsPath() 方法返回投影到楼梯基面的、表示梯段楼梯路径的曲线。GetFootprintBoundary() 方法返回同样也投射到楼梯基面的梯段边界曲线。

创建新梯段的三个 StairsRun 类静态方法, 已在 3.10.1 节楼梯中述及。

StairsRunType 类表示楼梯梯段类型。它包含许多关于踏板和踢面的属性以及其他与梯段相关的信息。代码 3-88 示例了如何获取楼梯图元中第一个梯段的梯段类型, 并连同类型名称一起显示踢面和踏板厚度。

代码 3-88: 获取 StairsRunType 信息

```
private void GetRunType (Stairs stairs)
{
    ICollection<ElementId> runIds = stairs.GetStairsRuns();

    ElementId firstRunId = runIds.First();

    StairsRun firstRun = stairs.Document.GetElement (firstRunId) as StairsRun;
    if (null != firstRun)
    {
        StairsRunType runType = stairs.Document.GetElement (firstRun.GetTypeId()) as StairsRunType;
        // Format landing type info for display
        string info = "Stairs Run Type:  " + runType.Name;
        info += "\nRiser Thickness:  " + runType.RiserThickness;
        info += "\nTread Thickness:  " + runType.TreadThickness;

        TaskDialog.Show ("Revit", info);
    }
}
```

## 2. 楼梯平台 (Landings)

楼梯平台由 StairsLanding 类表示。代码 3-89 示例了如何查找楼梯对象中每个平台的厚度。

代码 3-89: 使用 StairsLanding

```
private void GetStairLandings (Stairs stairs)
{
    ICollection<ElementId> landingIds = stairs.GetStairsLandings();
    string info = "Number of landings:  " + landingIds.Count;

    int landingIndex = 0;
```



```

foreach (ElementId landingId in landingIds)
{
    landingIndex++;
    StairsLanding landing = stairs.Document.GetElement (landingId) as StairsLanding;
    if (null != landing)
    {
        info += "\nThickness of Landing " + landingIndex + ": " + landing.Thickness;
    }
}

TaskDialog.Show ("Revit", info);
}

```

类似于楼梯梯段，楼梯平台包含 `GetStairsPath()` 方法，返回投影到楼梯基面的、代表平台段楼梯路径的曲线，以及 `GetFootprintBoundary()` 方法，返回同样投影到楼梯基面的平台边界曲线。同样类似于楼梯梯段，也有一个获取所有以平台为主体的支撑的方法。

`StairsLanding` 类含有在两个梯段之间新建平台的方法。已在 3.10.1 节楼梯中述及。

`StairsLandingType` 类表示 Revit API 中的楼梯平台类型。`StairsLandingType` 类只有两个特定属性，即 `IsMonolithic` 和 `Thickness`。若楼梯平台为整体式，则 `IsMonolithic` 为 `true`，`Thickness` 则表示楼梯平台厚度。

### 3. 楼梯构件连接 (StairsComponentConnection)

楼梯梯段和平台都有个 `GetConnections()` 方法，返回有关楼梯构件之间（梯段与梯段或梯段与平台）的连接信息。该方法可返回有关各个连接属性的 `StairsComponentConnection` 对象集合，其中包括连接类型（对于平台，为楼梯梯段始端或末端连接）和所连接楼梯构件的 ID。

#### 代码 3-90: 获取楼梯支撑

```

private void GetStairSupports (Stairs stairs)
{
    ICollection<ElementId> supportIds = stairs.GetStairSupports();
    string info = "Number of supports: " + supportIds.Count;

    int supportIndex = 0;
    foreach (ElementId supportId in supportIds)
    {
        supportIndex++;
        Element support = stairs.Document.GetElement (supportId);
        if (null != support)
        {
            info += "\nName of support " + supportIndex + ": " + support.Name;
        }
    }
}

```



```
TaskDialog.Show ("Revit", info);  
}
```

#### 4. 支撑 (Supports)

Revit API 未发布楼梯中用于支撑的类。获取楼梯、楼梯梯段或楼梯平台的支撑时，得到的都是一般的 Revit 图元。代码 3-90 可获取一个楼梯对象所有支撑的名称。

## 第 4 章 规程特有功能（Discipline-Specific Functionality）

### 4.1 Revit Architecture

本章包含特定于建筑规程的 API 功能，即房间相关功能（Element.Room、RoomTag 等）。

以下各节介绍房间有关的类、参数，以及 API 中如何使用房间类。

#### 1. 房间、面积、标记（Room、Area and Tags）

Room 类用于表示房间和图元，如房间明细表和面积平面。表 4-1 列出了 API 中不同的房间、面积及其相应标记的属性和创建函数。

表 4-1 房间、面积和标记关系

图元	类	类别	边界	位置	创建函数
平面视图中的房间	Room	OST_Rooms	若在封闭区域内，则有	LocationPoint	除 NewRoom（Phase）以外的 NewRoom（）
明细表视图中的房间	Room	OST_Rooms	null	null	NewRoom（Phase）
面积	Room	OST_Areas	总是有	LocationPoint	无
房间标记	RoomTag	OST_RoomTags		LocationPoint	Creation.Document.NewRoomTag（）
面积标记	FamilySymbol	OST_AreaTags		LocationPoint	不可

注：Room.Name 是房间名称和房间号的组合。使用 ROOM\_NAME 内建参数来获取房间的名称。在图 4-1 中，Room.Name 返回“Master Bedroom 2”。

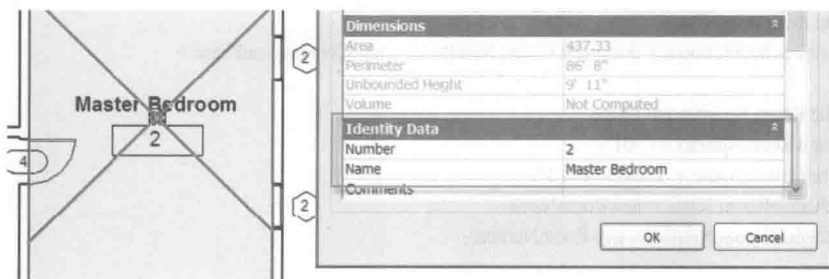


图 4-1 房间名称和房间号

请注意，作为一个注释图元，使用 RoomTag.View 可获得指定视图。绝对不要试图设置 RoomTag.Name 属性，因为名称是自动分配的；否则将引发异常。

#### 2. 创建房间（Creating a Room）

代码 4-1 说明了在指定标高点创建房间的最简单方法。

**代码 4-1: 创建房间**

```
Room CreateRoom (Autodesk.Revit.DB.Document document, Level level)
{
    // Create a UV structure which determines the room location
    UV roomLocation = new UV (0, 0);

    // Create a new room
    Room room = document.Create.NewRoom (level, roomLocation);
    if (null == room)
    {
        throw new Exception ("Create a new room failed.");
    }

    return room;
}
```

在房间明细表中可以创建房间，然后插入平面环回线。

- **Document.NewRoom (Phase)** 方法用于创建与任何指定位置无关的新房间，并将其插入现有明细表。确保在调用该方法之前，房间明细表已存在或创建一个特定阶段的房间明细表。
- **Document.NewRoom (Room room, PlanCircuit circuit)** 方法用于从明细表内的房间和 **PlanCircuit**，创建房间。
  - 所输入的房间必须仅存在于房间明细表中，也就是它在任何平面视图中都未显示。
  - 在调用该方法之后，具有相同名称和编号的模型房间被创建在 **PlanCircuit** 所在视图中。**PlanCircuit** 的详细信息，请参阅 4.1 节中的平面拓扑。

代码 4-2 说明了整个过程。

**代码 4-2: 创建并插入一个房间到平面环回线**

```
Room InsertNewRoomInPlanCircuit (Autodesk.Revit.DB.Document document, Level level, Phase newConstructionPhase)
{
    // create room using Phase
    Room newScheduleRoom = document.Create.NewRoom (newConstructionPhase);

    // set the Room Number and Name
    string newRoomNumber = "101";
    string newRoomName = "Class Room 1";
    newScheduleRoom.Name = newRoomName;
    newScheduleRoom.Number = newRoomNumber;

    // Get a PlanCircuit
    PlanCircuit planCircuit = null;
    // first get the plan topology for given level
    PlanTopology planTopology = document.get_PlanTopology (level);

    // Iterate circuits in this plan topology
    foreach (PlanCircuit circuit in planTopology.Circuits)
    {

```



```

// get the first circuit we find
if (null != circuit)
{
    planCircuit = circuit;
    break;
}

Room newRoom2 = null;
Transaction transaction = new Transaction (document, "Create Room");
if (transaction.Start() == TransactionStatus.Started)
{
    // The input room must exist only in the room schedule,
    // meaning that it does not display in any plan view.
    newRoom2 = document.Create.NewRoom (newScheduleRoom, planCircuit);
    // a model room with the same name and number is created in the
    // view where the PlanCircuit is located
    if (null != newRoom2)
    {
        // Give the user some information
        TaskDialog.Show ("Revit", "Room placed in Plan Circuit successfully.");
    }
    transaction.Commit();
}

return newRoom2;
}

```

房间创建并添加到某个位置后，用 `Room.Unplace()` 方法可从该位置删除它（但在项目中仍可使用）。然后，它可被放置在一个新位置上。

### 3. 房间边界 (Room Boundary)

房间含有边界，它创建一个房间所在位置的封闭区域。边界包含下列图元：

- 墙。
- 模型线。
- 柱。
- 屋顶。

### 4. 检索房间边界 (Retrieving Room Boundaries)

从基类方法 `SpatialElement.GetBoundarySegments()` 可得到围绕房间的边界。若房间不在一个封闭区域内或只存在于明细表中，此方法返回“null”。每个房间可能有若干区域，每个区域可能有若干区段，因此数据以 `BoundarySegment` 列表清单形式返回。

图 4-2 显示了在 Revit 用户界面中选择的房间边界。

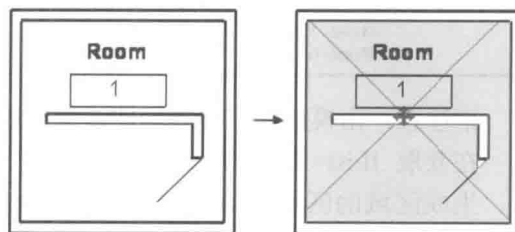


图 4-2 房间边界





区段列表的大小取决于封闭区域的拓扑。在一个区段与一个区段连接后，每个 BoundarySegment 列表会生成一个环回线或一条连续线。图 4-3 提供了几个例子。其中所有的墙都是 Room- Bounding，模型线类别是 OST\_AreaSeparationLines。如果某个图元不是房间边界，将被排除在生成边界的图元之外。

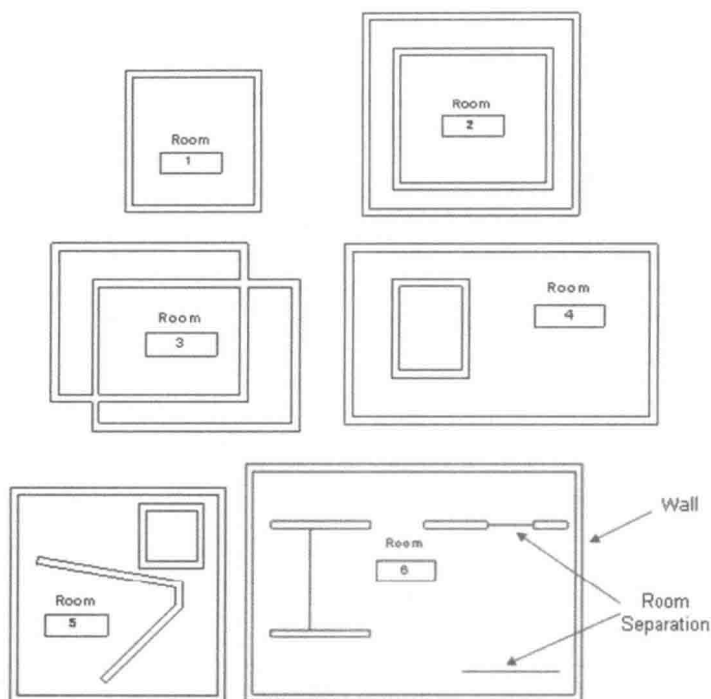


图 4-3 房间 1~房间 6

表 4-2 提供了对于上述房间 GetBoundarySegments().Size 的结果。

表 4-2

GetBoundarySegments().Size

房间	Room.GetBoundarySegments().Size
Room 1 Room 2 Room 3	1
Room 4	2
Room 5 Room 6	3

请注意，由模型线连接的墙被认为是连续线段。单一模型线会被忽略。

在获取 IList<IList<BoundarySegment>后，可通过迭代该列表获取 BoundarySegment。

生成区域的区段由 BoundarySegment 类表示，其 Element 属性返回符合下列条件的图元：

- 对于模型曲线图元，类别必须为 BuiltInCategory.OST\_AreaSeparationLines，意为它



表示房间分隔线。

- 对于其他图元，如墙、柱、屋顶，如果这些图元是某个房间的边界，房间边界参数（BuiltInParameter.WALL\_ATTR\_ROOM\_BOUNDING）必须为 true，见图 4-4。

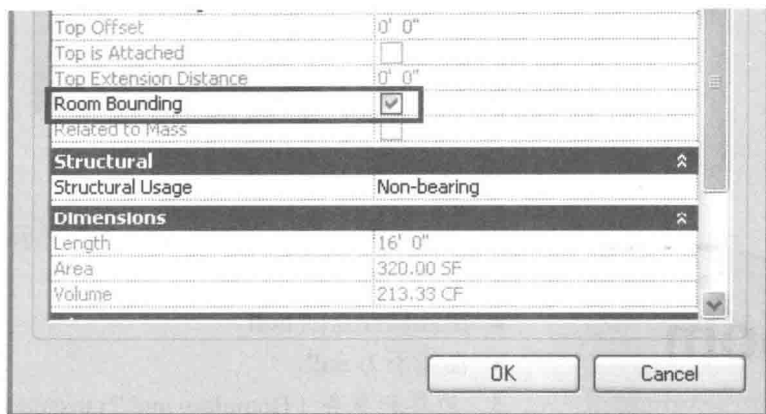


图 4-4 房间边界属性

通过 API 设置 WALL\_ATTR\_ROOM\_BOUNDING 内建参数，见代码 4-3。

#### 代码 4-3：设置房间边界

```
public void SetRoomBounding (Wall wall)
{
    Parameter parameter = wall.get_Parameter (BuiltInParameter.WALL_ATTR_ROOM_BOUNDING);
    parameter.Set (1);    //set "Room Bounding" to true
    parameter.Set (0);    //set "Room Bounding" to false
}
```

注意：图 4.5 和图 4.6 说明了屋顶如何形成房间的边界区段。图 4-5 显示了立面图中的标高 3。房间创建于标高 3 楼层平面图。图 4-6 显示了三维视图中的房子和房间边界。

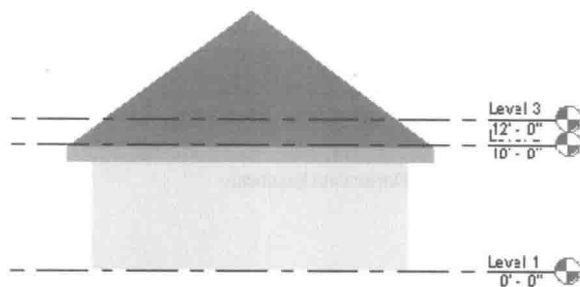


图 4-5 标高 3 视图中创建的房间

面积边界只能为 Area Boundary 类别的模型曲线（BuiltInCategory.OST\_AreaSchemeLines），而所显示的房间的边界可以是墙和其他图元。

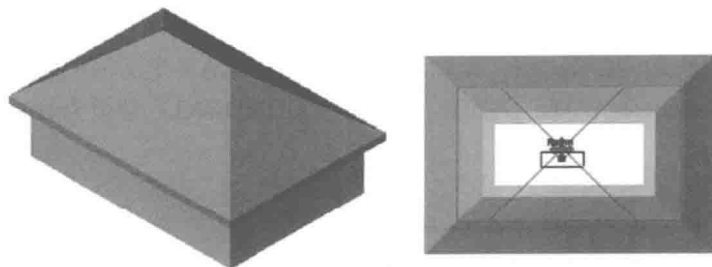


图 4-6 屋顶形成的房间边界

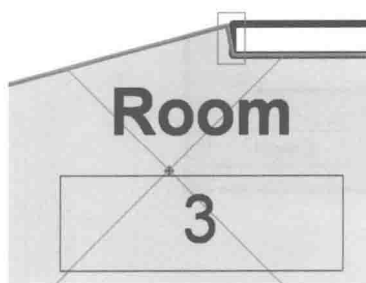


图 4-7 墙端边缘

若 BoundarySegment 对应于图 4-7 所示房间分隔和墙之间的曲线，则：

- 图元属性返回 null。
- 曲线不为 null。

#### 5. 边界和事务 (Boundary and Transaction)

用 API 创建某图元，如墙，然后调用 Room.GetBoundarySegments()，则该墙可以改变房间的边界。注意必须确保其改变后的数据更新。

图 4-8 展示了使用 Revit 平台 API 创建一面墙之后，房间如何改变。

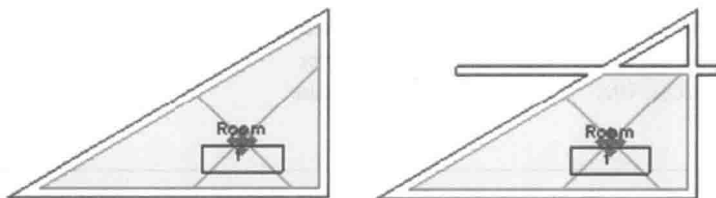


图 4-8 添加一面墙改变房间边界

要更新房间边界数据，请使用代码 4-4 中的事务处理方法。

#### 代码 4-4：使用事务处理来更新房间边界

```
public void UpdateRoomBoundary (UIApplication application, Room room, Level level)
{
    Document document = application.ActiveUIDocument.Document;

    //Get the size before creating a wall
    int size = room.GetBoundarySegments (new SpatialElementBoundaryOptions()).First().Count;
    string message = "Room boundary size before wall: " + size;

    //Prepare a line
    XYZ startPos = new XYZ (-10, 0, 0);
    XYZ endPos = new XYZ (10, 0, 0);
    Line line = Line.CreateBound (startPos, endPos);
```



```
//Create a new wall and enclose the creating into a single transaction
using (Transaction transaction = new Transaction ( document, "Create Wall"))
{
    if (transaction.Start() == TransactionStatus.Started)
    {
        Wall wall = Wall.Create (document, line, level.Id, false);
        if (null != wall)
        {
            if (TransactionStatus.Committed == transaction.Commit())
            {
                //Get the new size
                size = room.GetBoundarySegments (new SpatialElementBoundaryOptions()).First().Count;
                message += "\nRoom boundary size after wall: " + size;
                TaskDialog.Show ("Revit", message);
            }
        }
    }
    else
    {
        transaction.Rollback();
    }
}
}
```

有关详细信息，请参阅 5.2 节事务。

## 6. 平面拓扑 (Plan Topology)

房间所在楼层平面有一个由图元如墙和房间分隔线生成的拓扑结构。PlanTopology 和 PlanCircuit 类用于表示楼层平面拓扑，见图 4-9。

- 用 Level 可从 Document 对象获取 PlanTopology 对象。在每个平面视图中，都有一个对应各阶段的平面拓扑。
- 同样情况适用于边界区段，除非房间分隔线和 Room Bounding 参数为 true 的图元可成为 PlanCircuit 中的边线（边界）。

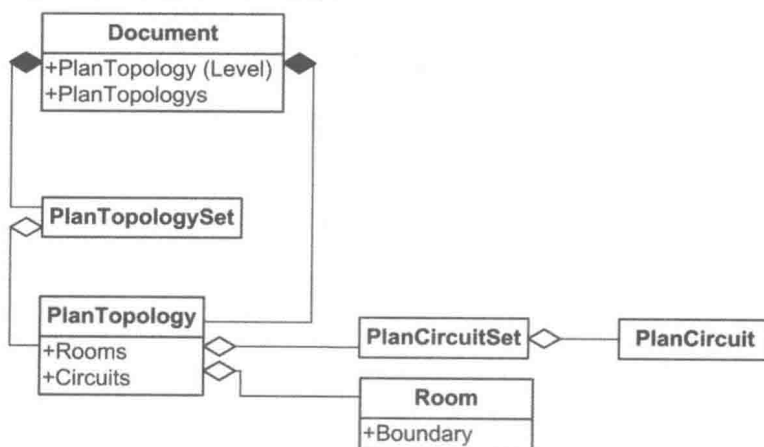


图 4-9 房间和平面拓扑图



PlanCircuit.SideNum 属性返回环回边线的数量, 而 SpatialElement.GetBoundarySegments() 返回一个 IList<IList<Autodesk.Revit.DB.BoundarySegment>, 其数目不同于环行边线的数量。图 4-10 和表 4-3 列出了房间边界和平面环回线的比较。

- 若墙上有一个分支, 则 SpatialElement.GetBoundarySegments() 将底墙识别为两面墙。
- 不管分支数量多少, PlanCircuit.SideNum 总是将图片中的底墙视为一个。

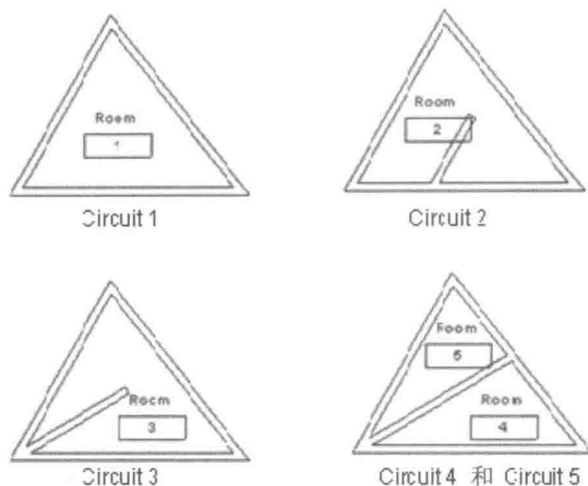


图 4-10 比较房间边界和平面环回线

表 4-3

房间边界和平面环回线比较

Circuit	Circuit.SideNum	IList<IList<Autodesk.Revit.DB.BoundarySegment>.Count for Room
Circuit 1	3	3 (Room1)
Circuit 2	4 + 2 = 6	4 + 3 = 7 (Room2)
Circuit 3	3 + 2 = 5	3 + 3 = 6 (Room3)
Circuit 4	3	3 (Room4)
Circuit 5	3	3 (Room5)

### 7. 房间和族实例 ( Room and FamilyInstance )

门、窗是与房间相关的特殊族实例。在此仅讨论门, 因为窗与门唯一的区别是, 窗没有反置的把手。

门有以下特点:

- 门可以独立于房间存在。
- 在 API 中 (也只有 API 中), Door 图元有两个附加属性, 其参考门的相反两个面域: ToRoom 和 FromRoom
- 若区域为房间, 则该属性的值将是一个 Room 图元。
- 若区域非房间, 则该属性返回 null。两个属性可以同时为 null。
- 门开向的一侧区域为 ToRoom, 另一侧的房间为 FromRoom。
- 两个属性都会随相应区域的变动而动态更新。



在图 4-11 中, 嵌入墙壁的五扇门均无翻转饰面。表 4-4 列出了各扇门的 FromRoom、ToRoom 和 Room 属性。Room 属性属于所有族实例。

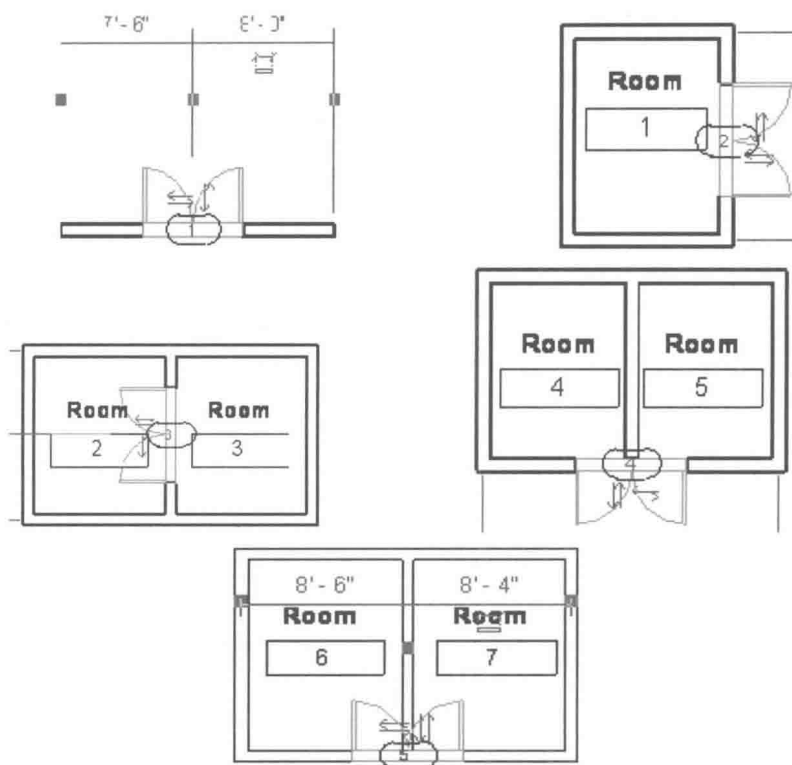


图 4-11 门 1~门 5

表 4-4

门的属性

门	FromRoom	ToRoom	Room
Door 1	null	null	null
Door 2	Room 1	null	null
Door 3	Room 3	Room 2	Room 2
Door 4	Room 4	null	null
Door 5	null	Room 6	Room 6

所有族实例都有 Room 属性, 它是实例在最终项目阶段所位于的房间。门和窗面向房间内。通过反置门或窗的开向, 或调用 FamilyInstance.FlipFromToRoom(), 可转换房间。对于其他种类的实例, 如梁、柱, Room 是与实例有相同边界的房间。

代码 4-5 说明了如何从族实例获取 Room。有必要检查其结果是否为 null。

#### 代码 4-5: 由族实例获取房间

```
public void GetRoomInfo (FamilyInstance familyInstance)
{
```



```

Room room = familyInstance.Room;
room = familyInstance.FromRoom; //for door and window family only
room = familyInstance.ToRoom;   //for door and window family only
if ( null != room )
{
    //use the room...
}
}

```

#### 8. 其他房间属性 ( Other Room Properties )

Room 类有几个其他属性可用于获取有关对象的信息, 见图 4-12。房间有一些只读的尺寸属性如下:

- Area (面积)。
- Perimeter (周长)。
- UnboundedHeight (房间标示高度)。
- Volume (体积)。
- ClosedShell (闭合壳)。

Instance Parameters - Control selected or to-be-created instance

Parameter	Value
<b>Constraints</b>	
Level	Level 1
Upper Limit	Level 1
Limit Offset	8' 0"
Base Offset	0' 0"
<b>Dimensions</b>	
Area	475.28 SF
Perimeter	90' 3"
Unbounded Height	8' 0"
Volume	Not Computed
<b>Identity Data</b>	
Number	1
Name	Living Room
Comments	

图 4-12 Room 类的尺寸属性

代码 4-6 显示了所选房间的尺寸信息。要注意的是, 体积计算设置必须为“启用”, 否则房间体积返回值为 0。

#### 代码 4-6: 获取房间尺寸

```

public void GetRoomDimensions ( Document doc, Room room )
{
    String roominfo = "Room dimensions: \n";
    // turn on volume calculations:
    VolumeCalculationOptions options = new VolumeCalculationOptions();
    options.VolumeComputationEnable = true;
    doc.Settings.VolumeCalculationSetting.VolumeCalculationOptions = options;
}

```



```
roominfo += "Vol: " + room.Volume + "\n";  
roominfo += "Area: " + room.Area + "\n";  
roominfo += "Perimeter: " + room.Perimeter + "\n";  
roominfo += "Unbounded height: " + room.UnboundedHeight + "\n";  
TaskDialog.Show ("Revit", roominfo);  
}
```

Room（或 Space）的 ClosedShell 属性，是由房间的开敞空间边界（墙壁、楼板、天花板、屋顶和边界线）形成的几何形状。如果开发人员需要检查房间与模型中其他具体图元是否相交，该属性是很有用的，例如，查看某个图元的部分或全部是否位于房间内。有关示例，请参见包含在 Revit SDK 中的 RoofsRooms 示例应用程序，ClosedShell 用来检查房间是否为垂直无界。

此外，可用以下属性获取或设置房间的底部偏移和高度偏移：

- BaseOffset。
- LimitOffset。

还可以用 UpperLimit 属性获取或设置定义房间上限标高。

## 4.2 Revit Structure

以下各节讨论只存在于 Revit Structure 产品中的一些 API 特性：

- 结构模型图元：探讨那些仅存在于 Revit Structure 产品中的特定图元及其属性。
- 分析模型：探讨分析模型相关的类，如 AnalyticalModel、RigidLink 和 Analytical ModelSupport。
- 分析链接：讨论在分析梁和柱之间创建新的分析链接。
- 荷载：讨论荷载设置及三种荷载。
- 分析链接建议：为需要将 Revit Structure 产品链接到某些结构分析应用程序的 API 用户提供的一些建议。

本章包括一些进阶专题。如果您还不太熟悉 Revit 平台 API，请首先阅读基础章节，如入门、图元概要、参数等。

### 4.2.1 结构模型图元（Structural Model Elements）

结构模型图元，从字面上看，是支持结构的图元，如柱、钢筋、桁架等。本节讨论如何操控这些图元。

#### 1. 柱、梁和支撑（Column, Beam and Brace）

结构柱、梁和支撑图元没有诸如 StructuralColumn 等特指的类，但它们可以用 FamilyInstance 类形式来表示。

请注意，虽然目前 API 中不存在 StructuralColumn、StructuralBeam 和 StructuralBrace 类，但本章用它们来表示对应于结构柱、梁和支撑的 FamilyInstance 对象。

虽然结构柱、梁和支撑都是 API 中的族实例对象，但 StructuralType 属性可对其进行区分，见代码 4-7。



**代码 4-7: 区分柱、梁和支撑**

```
public void GetStructuralType (FamilyInstance familyInstance)
{
    string message = "";
    switch (familyInstance.StructuralType)
    {
        case StructuralType.Beam:
            message = "FamilyInstance is a beam.";
            break;
        case StructuralType.Brace:
            message = "FamilyInstance is a brace.";
            break;
        case StructuralType.Column:
            message = "FamilyInstance is a column.";
            break;
        case StructuralType.Footing:
            message = "FamilyInstance is a footing.";
            break;
        default:
            message = "FamilyInstance is non-structural or unknown framing.";
            break;
    }
    TaskDialog.Show ("Revit", message);
}
```

代码 4-8 可以用类别过滤出对应于结构柱、梁和支撑的 `FamilySymbol` 对象。结构梁和支撑的类别是 `BuiltInCategory.OST_StructuralFraming`。

**代码 4-8: 使用 `BuiltInCategory.OST_StructuralFraming`**

```
public void GetBeamAndColumnSymbols (Document document)
{
    FamilySymbolSet columnTypes = new FamilySymbolSet();
    FamilySymbolSet framingTypes = new FamilySymbolSet();
    FilteredElementCollector collector = new FilteredElementCollector (document);
    ICollection<Element> elements = collector.OfClass (typeof (Family)) .ToElements();

    foreach (Element element in elements)
    {
        Family tmpFamily = element as Family;
        Category category = tmpFamily.FamilyCategory;
        if (null != category)
        {
            if ((int) BuiltInCategory.OST_StructuralColumns == category.Id.IntegerValue)
            {
                foreach (FamilySymbol tmpSymbol in tmpFamily.Symbols)
                {
                    columnTypes.Insert (tmpSymbol);
                }
            }
            else if ((int) BuiltInCategory.OST_StructuralFraming == category.Id.IntegerValue)
```



```

    {
        foreach (FamilySymbol tmpSymbol in tmpFamily.Symbols)
        {
            framingTypes.Insert (tmpSymbol);
        }
    }
}

string message = "Column Types: ";
FamilySymbolSetIterator fsItr = columnTypes.ForwardIterator();
fsItr.Reset();
while (fsItr.MoveNext())
{
    FamilySymbol familySybmol = fsItr.Current as FamilySymbol;
    message += "\n" + familySybmol.Name;
}
TaskDialog.Show ("Revit", message);
}

```

可以用 `FamilyInstance.ExtensionUtility` 属性获取和设置梁缩进属性。如果此属性返回 `null`, 则梁缩进无法修改。

## 2. 区域钢筋和路径钢筋 (AreaReinforcement and PathReinforcement)

通过调用 `GetCurveElementIds()` 方法返回一个代表区域钢筋曲线的 `ElementIds` 的 `IList` (链表接口), 可找出 `AreaReinforcement` 的 `AreaReinforcementCurves`。在图 4-13 的 Revit 用户界面中, 编辑模式下区域钢筋曲线为紫色 (选中时为红色)。从图元属性对话框中可以看到区域 `AreaReinforcementCurve` 参数。如 `Hook Types` 和 `Orientation` 参数, 只有在 `Override Area Reinforcement Setting` (重写区域钢筋设置) 参数为 `true` 时, 这些参数才是可编辑的。

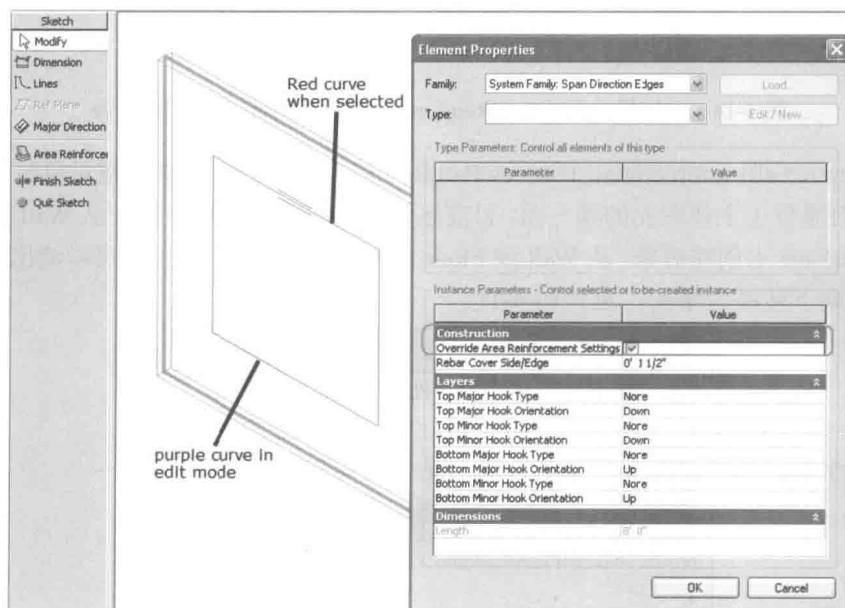


图 4-13 编辑模式下的区域钢筋曲线



使用 `NewAreaReinforcement()` 方法和 `AreaReinforcement.Direction` 属性创建一个新的区域钢筋时, 可以设置区域钢筋的 Major Direction (最终 XYZ 方向输入参数), 见代码 4-9。

**代码 4-9: NewAreaReinforcement()**

```
public AreaReinforcement NewAreaReinforcement (
    Element host, CurveArray curves, XYZ direction);
```

虽然 `AreaReinforcement.GetCurveElementIds()` 方法返回一个代表区域钢筋曲线的 `ElementIds` 集, 其中有个属性返回一个 `Curve`, 但 `PathReinforcement.GetCurveElementIds()` 方法返回一个代表 `ModelCurves` 的 `ElementIds` 的集合。除了通过使用 `NewPathReinforcement()` 方法 (最终输入参数), 没有其他方法可以翻转路径钢筋, 参见代码 4-10 和图 4-14。

**代码 4-10: NewPathReinforcement()**

```
public PathReinforcement NewPathReinforcement (
    Element host, CurveArray curves, bool flip);
```

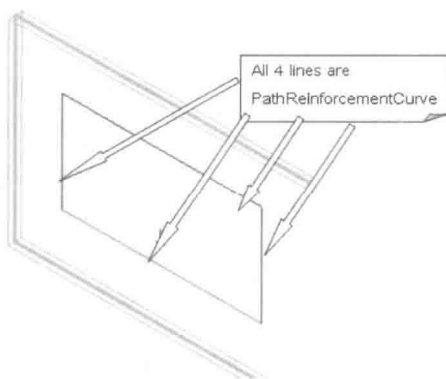


图 4-14 编辑模式下的 `PathReinforcement.PathReinforcementCurve`

使用 `NewAreaReinforcement()` 和 `NewPathReinforcement()` 方法来创建对象时, 开发人员必须决定该对象位于主体图元的哪一面。目前区域钢筋和路径钢筋只能在从 `Wall` 或 `Floor` 检索到的 `PlanarFace` 上创建对象。从 `Wall` 或 `Floor` 几何体上去除表面之后, 可以滤出平面如下:

- 将面向下转换为平面, 见代码 4-11。

**代码 4-11: 将 Face 向下转换为 PlanarFace**

```
PlanarFace planarFace = face as PlanarFace;
```

- 若为平面, 则再获取其法线和原点, 见代码 4-12。

**代码 4-12: 获取法线和原点**

```
private void GetPlanarFaceInfo (Face face)
{
    PlanarFace planarFace = face as PlanarFace;
    if (null != planarFace)
```



```

{
    XYZ origin = planarFace.Origin;
    XYZ normal = planarFace.Normal;
    XYZ vector = planarFace.get_Vector (0);
}
}

```

- 根据法线和原点，滤出合适的面，例如：
  - 对垂向的常规墙，根据以下因素放置面：
  - 面为垂直面；（normal.Z == 0.0）
  - 反方向平行面：
  - （normal1.X = - normal2.X； normal1.Y = - normal2.Y）
  - 法线必须平行于定位线。
  - 对于没有坡度的楼板，考虑以下因素：
  - 面为水平面；（normal.X == 0.0 && normal.Y == 0.0）
  - 判断顶面和底面；（根据 normal.Z 区别这两个面）

有关检索图元几何的更多细节，请参阅 3.7 节几何。

请注意，项目范围内的有关区域和路径钢筋的设置，可从 ReinforcementSettings 类访问。目前，唯一可用的设置是 HostStructuralRebar 属性。

### 3. 梁系统（BeamSystem）

梁系统支持完全访问和编辑，以获取和设置它的所有属性，如 BeamSystemType、BeamType、Direction 和 Level。BeamSystem.Direction 不限于某一条边线。它可被设置为 BeamSystem 同一平面上的任意坐标。

请注意，在用户界面或通过 API 更改 Elevation 属性后，无法更改其结构梁分析模型。如图 4-15 所示，梁系统高程更改为 10 英尺后，分析模型仍留在原位置上。

### 4. 桁架（Truss）

Truss 类表示 Revit 中所有类型的桁架。TrussType 属性表示桁架类型，参见代码 4-13。

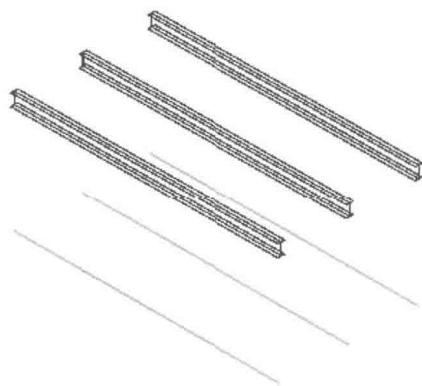


图 4-15 更改梁系统高程

#### 代码 4-13：在两根柱上创建桁架

```

Truss CreateTruss (Autodesk.Revit.DB.Document document,
    FamilyInstance column1, FamilyInstance column2)
{
    Truss truss = null;
    using (Transaction transaction = new Transaction (document, "Add Truss"))
    {
        if (transaction.Start() == TransactionStatus.Started)

```



```
{
    //sketchPlane
    XYZ origin = new XYZ (0, 0, 0);
    XYZ xDirection = new XYZ (1, 0, 0);
    XYZ yDirection = new XYZ (0, 1, 0);
    Plane plane = document.Application.Create.NewPlane (xDirection, yDirection, origin);
    SketchPlane sketchPlane = SketchPlane.Create (document, plane);

    //new base Line - use line that spans two selected columns
    AnalyticalModel frame1 = column1.GetAnalyticalModel() as AnalyticalModel;
    XYZ centerPoint1 = (frame1.GetCurve() as Line) .GetEndPoint (0);

    AnalyticalModel frame2 = column2.GetAnalyticalModel() as AnalyticalModel;
    XYZ centerPoint2 = (frame2.GetCurve() as Line) .GetEndPoint (0);

    XYZ startPoint = new XYZ (centerPoint1.X, centerPoint1.Y, 0);
    XYZ endPoint = new XYZ (centerPoint2.X, centerPoint2.Y, 0);
    Autodesk.Revit.DB.Line baseLine = null;

    try
    {
        baseLine = Line.CreateBound (startPoint, endPoint);
    }
    catch (System.ArgumentException)
    {
        throw new Exception ("Selected columns are too close to create truss.");
    }

    // use the active view for where the truss's tag will be placed; View used in
    // NewTruss should be plan or elevation view parallel to the truss's base line
    Autodesk.Revit.DB.View view = document.ActiveView;

    // Get a truss type for the truss
    FilteredElementCollector collector = new FilteredElementCollector (document);
    collector.OfClass (typeof (FamilySymbol));
    collector.OfCategory (BuiltInCategory.OST_Truss);

    TrussType trussType = collector.FirstElement() as TrussType;

    if (null != trussType)
    {
        truss = Truss.Create (document, trussType.Id, sketchPlane.Id, baseLine);
        transaction.Commit();
    }
    else
    {
        transaction.Rollback();
        throw new Exception ("No truss types found in document.");
    }
}
```



```
    }  
    }  
  
    return truss;  
}
```

5. 钢筋 (Rebar)

Rebar 类表示用于适合配筋图元的钢筋，如混凝土梁、柱、板或基础。可以使用表 4- 5 三种静态 Rebar 方法之一来创建钢筋对象。

表 4-5 创建钢筋对象的静态 Rebar 方法

名 称	说 明
<pre>public static Rebar Rebar.CreateFromCurves (     Document doc,     RebarStyle style,     RebarBarType rebarType,     RebarHookType startHook,     RebarHookType endHook,     Element host,     XYZ norm,     IList&lt;Curve&gt; curves,     RebarHookOrientation startHookOrient,     RebarHookOrientation endHookOrient,     bool useExistingShapeIfPossible,     bool createNewShape );</pre>	在项目中创建 Rebar 图元的新实例。所有曲线必须属于由法线和原点所定义的平面
<pre>public static Rebar Rebar.CreateFromRebarShape (     Document doc,     RebarShape rebarShape,     RebarBarType rebarType,     Element host,     XYZ origin,     XYZ xVec,     XYZ yVec );</pre>	创建新的 Rebar 作为钢筋形状的实例。实例会有来自 RebarShape 的默认形状参数，它根据形状定义中的形状边界框来定位。弯钩在计算边界框之前被从形状中删除。如果文件中能找到合适的弯钩，可任意分配
<pre>public static Rebar Rebar.CreateFromCurvesAndShape (     Document doc,     RebarShape rebarShape,     RebarBarType rebarType,     RebarHookType startHook,     RebarHookType endHook,     Element host,     XYZ norm,     IList&lt;Curve&gt; curves,     RebarHookOrientation startHookOrient,     RebarHookOrientation endHookOrient );</pre>	在项目中创建 Rebar 图元的新实例。实例会有来自 RebarShape 的默认形状参数。所有曲线必须属于由法线和原点所定义的平面

第一种方法由曲线数组创建钢筋来绘制钢筋，而第二种方法根据钢筋形状和位置来创建 Rebar 对象，第三种方法根据 RebarShape 由曲线数组创建钢筋。

使用 CreateFromCurves()或 CreateFromCurvesAndShape()方法时，参数 RebarBarType 和 RebarHookType 可从 Document 的 RebarBarTypes 和 RebarHookTypes 属性中获取。



代码 4-14 说明了如何创建具有特定布局的 Rebar。

**代码 4-14: 创建具有特定布局的钢筋**

```
Rebar CreateRebar (Autodesk.Revit.DB.Document document, FamilyInstance column, RebarBarType barType, RebarHookType
hookType)
{
    // Define the rebar geometry information - Line rebar
    LocationPoint location = column.Location as LocationPoint;
    XYZ origin = location.Point;
    XYZ normal = new XYZ (1, 0, 0);
    XYZ rebarLineEnd = new XYZ (origin.X, origin.Y, origin.Z + 9);
    Line rebarLine = Line.CreateBound (origin, rebarLineEnd);

    // Create the line rebar
    IList<Curve> curves = new List<Curve>( );
    curves.Add (rebarLine);

    Rebar rebar = Rebar.CreateFromCurves (document, Autodesk.Revit.DB.Structure.RebarStyle.Standard, barType, hookType,
hookType,
        column, origin, curves, RebarHookOrientation.Right, RebarHookOrientation.Left, true, true);

    if (null != rebar)
    {
        // set specific layout for new rebar
        Parameter paramLayout = rebar.get_Parameter (BuiltInParameter.REBAR_ELEM_LAYOUT_RULE);
        paramLayout.Set (1); // 1 = Fixed Number
        Parameter paramNum = rebar.get_Parameter (BuiltInParameter.REBAR_ELEM_QUANTITY_OF_BARS);
        paramNum.Set (10);
        rebar.ArrayLength = 1.5;
    }

    return rebar;
}
```

注：创建钢筋图元的更多示例，请参见 Revit SDK 中 Reinforcement and NewRebar 示例应用程序。

表 4-6 列出了参数 REBAR\_ELEM\_LAYOUT\_RULE 的整型值。

**表 4-6 钢筋布置规则**

值	0	1	2	3	4
说明	无	固定根数	最大间距	间隔数量	最小净距

Rebar. ScaleToBox()方法提供了一种方式，来同时设置所有形状参数。类似于用户界面中放置钢筋。

**6. 钢筋主体数据及钢筋保护层类型 ( RebarHostData and RebarCoverType )**

净保护层与有效钢筋主体的单个面相关。通过 Autodesk.Revit. Elements. RebarHostData 对象，开发人员可以访问主体的保护层设置。还支持通过参数来访问同一设置，这是个更



简单但功能不怎么强的访问途径。

保护层由指定偏移距离来定义，模型化为 Autodesk.Revit.DB.Structure.RebarCoverType 图元。

### 7. 边界条件 (BoundaryConditions)

有三种类型的边界条件：

- Point (点)。
- Curve (曲线)。
- Area (区域)。

使用代码 4-15 来检索类型和相关的几何信息。

**代码 4-15：获取边界条件和几何信息**

```
public void GetInfo_BoundarySegment (Room room)
{
    IList<IList<Autodesk.Revit.DB.BoundarySegment> segments = room.GetBoundarySegments ( new SpatialElement
    BoundaryOptions());

    if (null != segments) //the room may not be bound
    {
        string message = "BoundarySegment";
        foreach (IList<Autodesk.Revit.DB.BoundarySegment> segmentList in segments)
        {
            foreach (Autodesk.Revit.DB.BoundarySegment boundarySegment in segmentList)
            {
                // Get curve start point
                message += "\nCurve start point: (" + boundarySegment.Curve.GetEndPoint (0).X + ", "
                    + boundarySegment.Curve.GetEndPoint (0).Y + ", " +
                    boundarySegment.Curve.GetEndPoint (0).Z + ") ";

                // Get curve end point
                message += "\nCurve end point: (" + boundarySegment.Curve.GetEndPoint (1).X + ", "
                    + boundarySegment.Curve.GetEndPoint (1).Y + ", " +
                    boundarySegment.Curve.GetEndPoint (1).Z + ") ";

                // Get document path name
                message += "\nDocument path name: " + boundarySegment.Document.PathName;
                // Get boundary segment element name
                if (boundarySegment.Element != null)
                {
                    message += "\nElement name: " + boundarySegment.Element.Name;
                }
            }
        }

        TaskDialog.Show ("Revit", message);
    }
}
```

### 8. 其他结构图元 (Other Structural Elements)

Revit Architecture 和 Revit Structure 软件中还有一些 Element 派生类。本节中介绍了特





定于 Revit Structure 的方法。有关这些类的更多信息，请参阅 3.1 节墙、楼板、屋顶和洞口以及 3.2 节族实例中的相应部分。

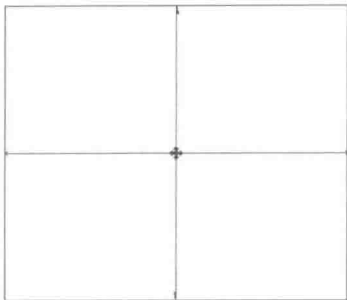


图 4-16 板跨方向

### 9. 板 ( Slab )

板 ( 结构楼板 ) 和板式基础都是由 Floor 类表示，并由 IsFoundationSlab 属性区分。

板跨方向由 API 的 IndependentTag 类表示，见图 4-16。

当使用 NewSlab ( ) 创建 Slab 时，并不会自动创建 Span Directions。也没有直接的方法来创建它们。

Slab 复合结构层的 Structural Deck 属性由以下属性公开：

- CompoundStructuralLayer.DeckUsage。
- DeckProfile。

图 4-17 对话框概括了这两个属性。

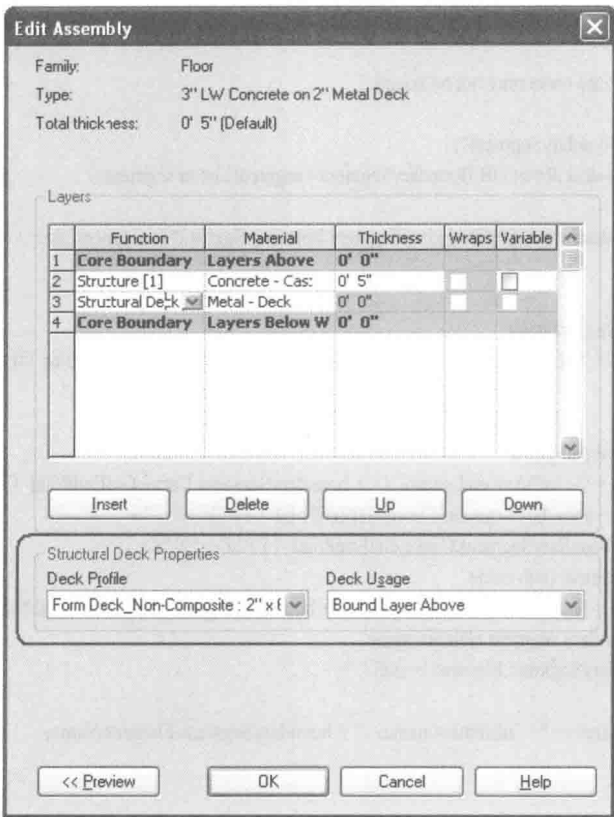


图 4-17 楼板的复合结构层属性

### 4.2.2 分析模型 ( Analytical Model )

在 Revit Structure 中，分析模型是结构物理模型的设计描述。

以下结构图元具有结构构件分析模型：

- 结构柱。



- 结构框架图元（如梁和支撑）。
- 结构楼板。
- 结构独立基础。
- 结构墙。

图元的分析模型可以用 `GetAnalyticalModel()` 方法来获取。请注意，新创建结构图元的分析模型是不可用的，直到发生重生成。是否存在分析模型取决于图元的族。若 `AnalyticalModel` 值不适用于图元的族，则 `GetAnalyticalModel()` 方法返回 `null`。在使用该类之前，请检查它的值。分析模型包含以下信息：

- 关于分析的图元位置。
- 参数信息，包括投影、硬点、近似，以及刚性连接。
- 支撑信息。
- 对齐信息，包括手动对齐和自动对齐。
- 分析偏移。

#### 1. 分析位置（Analytical Location）

根据分析模型对应的图元类型，与分析相关的图元位置可以通过三种方法之一来获得：`GetPoint()`，`GetCurve()`或 `GetCurves()`。

请注意，从这些方法中检索的曲线不含 `Reference` 属性设置。因此，它们不能用作属性，如 `Curve.EndPointReference`。但通过构建一个包含必要信息的 `AnalyticalModelSelector` 对象，可以获取曲线的参照及其端点，见代码 4-16。

**代码 4-16：获取分析曲线的参照**

```
public bool GetReferenceData (FamilyInstance familyInst)
{
    AnalyticalModel analyticalModelFrame = familyInst.GetAnalyticalModel();
    Curve analyticalCurve = analyticalModelFrame.GetCurve();
    if (null != analyticalCurve)
    {
        // test the stable reference to the curve.
        AnalyticalModelSelector amSelector = new AnalyticalModelSelector (analyticalCurve);
        amSelector.CurveSelector = AnalyticalCurveSelector.WholeCurve;
        Reference curveReference = analyticalModelFrame.GetReference (amSelector);

        // test the stable reference to the start point of the curve
        amSelector.CurveSelector = AnalyticalCurveSelector.StartPoint;
        Reference startPointReference = analyticalModelFrame.GetReference (amSelector);

        // test the stable reference to the end point of the curve
        amSelector.CurveSelector = AnalyticalCurveSelector.EndPoint;
        Reference endPointReference = analyticalModelFrame.GetReference (amSelector);
    }

    return true;
}
```



(1) `GetPoint()`。如果分析模型可以用单个点来表示 (如 `Structural Footing`)，则此方法将返回该点。否则将引发 `Autodesk. Revit. Exceptions.InapplicableDataException`。`IsSinglePoint()`方法可用于确定分析模型是否可用单个点来表示。

代码 4-17 演示了如何获取结构独立基础的分析位置。

#### 代码 4-17: 获取结构独立基础的位置

```
// retrieve and iterate current selected element
UIDocument uidoc = commandData.Application.ActiveUIDocument;
ElementSet selection = uidoc.Selection.Elements;
foreach (Element e in selection)
{
    // if the element is structural footing
    FamilyInstance familyInst = e as FamilyInstance;
    if (null != familyInst && familyInst.StructuralType == StructuralType.Footing)
    {
        AnalyticalModel model = familyInst.GetAnalyticalModel();
        // structural footing should be expressable as a single point
        if (model.IsSinglePoint() == true)
        {
            XYZ analyticalLocationPoint = model.GetPoint();
        }
    }
}
```

(2) `GetCurve()`。如果分析模型可以用单一曲线来表示 (如结构柱或结构框架)，此方法将返回其曲线。否则将引发 `Autodesk. Revit. Exceptions.InapplicableDataException`。`IsSingleCurve`方法可用于确定分析模型是否可以用单一曲线来表示，参见代码 4-18。

#### 代码 4-18: 获取结构柱曲线

```
public void GetColumnCurve (FamilyInstance familyInst)
{
    // get AnalyticalModel from structural column
    if (familyInst.StructuralType == StructuralType.Column)
    {
        AnalyticalModel modelColumn = familyInst.GetAnalyticalModel();
        // column should be represented by a single curve
        if (modelColumn.IsSingleCurve() == true)
        {
            Curve columnCurve = modelColumn.GetCurve();
        }
    }
}
```

(3) `GetCurves()`。要获取由多条曲线定义的分析模型的曲线 (如结构墙) 时，使用此方法，也可用于所有情况。如果分析模型可以用单一曲线表示，则此方法将返回只包含一



条曲线的列表。如果分析模型可以用单个点表示，则此方法将返回包含该点的、几乎是 0 长度的曲线。该方法以一个 `AnalyticalCurveType` 枚举作为参数。可能的值为：

- **RawCurves**: 所生成的基础分析模型曲线。
  - **ActiveCurves**: 显示在屏幕上的曲线（不包括刚性连接）。
  - **ApproximatedCurves**: 由多个直线段组成的近似曲线。
- 还可以得到刚性连接相关的下列值。更多内容，请参阅本章的刚性连接部分。
- **RigidLinkHead**: 梁的 0 端（首端）刚性连接。
  - **RigidLinkTail**: 梁的 1 端（尾端）刚性连接。
  - **AllRigidLinks**: 完全刚性连接。梁的 0 端（首端）刚性连接为第一项，梁的 1 端（尾端）刚性连接为最后一项。

代码 4-19 演示了如何使用结构墙分析模型。

**代码 4-19: 获取结构墙曲线**

```
// retrieve and iterate current selected element
UIDocument uidoc = commandData.Application.ActiveUIDocument;
ElementSet selection = uidoc.Selection.Elements;
foreach (Element e in selection)
{
    Wall aWall = e as Wall;
    if (null != aWall)
    {
        // get AnalyticalModelWall from Structural Wall
        AnalyticalModel modelWall =
            aWall.GetAnalyticalModel() as AnalyticalModel;
        if (null == modelWall)
        {
            // Architecture wall doesn't have analytical model
            continue;
        }
        // get analytical information
        int modelCurveNum = modelWall.GetCurves (AnalyticalCurveType.ActiveCurves) .Count;
        IList<AnalyticalModelSupport> supportList = new List<AnalyticalModelSupport>();
        supportList = modelWall.GetAnalyticalModelSupports();
        int supportInfoNum = supportList.Count;
    }
}
```

## 2. 参数信息 (Parameter Information)

分析模型支持参数信息访问，如刚性连接、投影和近似。

(1) 刚性连接 (Rigid Links)。刚性连接将梁的分析模型连接到柱的分析模型。使用 `CanHaveRigidLinksMethod()` 和 `AnalyticalModel.RigidLinksOption` 属性以确定刚性连接是否适用于该分析模型。此外，还可以使用 `HasRigidLinksWith()` 以确定分析模型是否已与特定图元刚性连接，参见图 4-18。

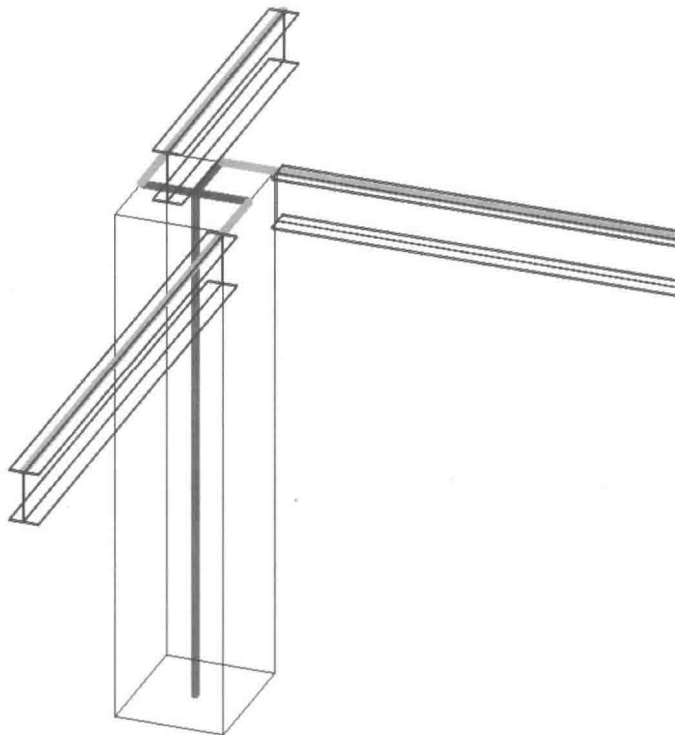


图 4-18 刚性连接

通过指定 `AnalyticalCurveType` 选项 `RigidLinkHead` 和 `RigidLinkTail`, 使用 `AnalyticalModel.GetCurves()` 方法, 或者对 `AnalyticalModelSelector` 对象使用 `AnalyticalModel.GetRigidLink()`, 可以检索到端部连接。

对于结构梁, 分析模型方法 `GetCurve()` 和 `GetCurves()` 之间的区别是, `GetCurves()` 既包含单一曲线, 还包含结构梁的刚性连接曲线 (如果有)。传递 `AnalyticalCurveType.RigidLinkHead` 或 `AnalyticalCurveType.RigidLinkTail` 枚举值到 `GetCurves()` 方法, 可获取梁首端或尾端的刚性连接。

虽然不能直接创建刚性连接, 因为它不是个独立对象, 但可以在梁和/或柱的分析模型上使用 `RigidLinksOption` 属性来创建它。梁的刚性连接选项会覆盖柱的相应选项。

对于结构梁, `RigidLinksOption` 属性可以是下列值:

- `AnalyticalRigidLinksOption.Enabled`: 生成刚性连接。
- `AnalyticalRigidLinksOption.Disabled`: 不生成刚性连接。
- `AnalyticalRigidLinksOption.FromColumn`: 可生成刚性连接, 取决于结构柱的值。

对于结构柱, `RigidLinksOption` 属性可为下列值:

- `AnalyticalRigidLinksOption.Enabled`: 将会生成刚性连接。
- `AnalyticalRigidLinksOption.Disabled`: 不生成刚性连接。

注: 除设置合适的值以外, 还必须交叠图元才可创建刚性连接。

(2) 分析投影 (`Analytical Projection`)。使用 `GetAnalyticalProjectionType()` 和 `SetAnalytical`



**ProjectionType()**方法，可以检索和设置分析模型的水平和垂直投影参数。这些方法将 **AnalyticalDirection** 作为输入。**SetAnalytical ProjectionDatumPlane()**还可用来设置指定投影到特定 **Datum Plane**，当设置到基准平面时，**GetAnalyticalProjectionDatumPlane()**会检索分析投影。

(3) 近似 (**Approximation**)。当分析模型由曲线而不是直线来定义 (即曲梁) 时，近似 (由直线段组成) 可能更为可取。分析模型有几个曲线近似相关方法。如果 **CanApproximate()** 返回 **true**，可使用 **Approximate()**方法在非近似 (曲线) 分析模型和近似 (仅由直线段组成) 分析模型之间切换。在切换到近似分析模型后，可使用 **GetCurves()**得到近似曲线的各直线段。

近似是建立在近似误差值 [**GetApproximationDeviation()**] 和使用硬点参数 [**UsesHardPoints()**] 的基础上。同样，这些值都具有相应的 **Set** 方法。近似误差限制光滑曲线与由“近似”产生的线段之间的差距。硬点是曲梁上接触其他结构图元的位置。当将此参数设置为 **true** 时，它强制分段分析模型在附着于曲梁的构件端部产生节点。

(4) 分析为 (**AnalyzeAs**)。通过分析模型可以检索和设置 **Analyze As** 参数。此参数指示分析程序该如何分析图元，或图元是否为 **NotForAnalysis**。由于 **GetAnalyzeAs()**和 **SetAnalyzeAs()**所使用的 **AnalyzeAs** 枚举包含用于各种类型图元的枚举值，因此并非所有的值都适用于所有的分析模型。可使用 **IsAnalyzeAsValid()**方法来确定某个特定值是否适用于某分析模型。

### 3. 手动对齐 (**Manual Adjustment**)

还可以将结构构件分析模型的几何体和那些与其连接的相关图元对齐 (假设 **SupportsManualAdjustment** 方法返回 **true**)。使用 **AnalyticalModel. ManuallyAdjust()**方法将分析模型与其他相关图元对齐，参见代码 4-20。

**代码 4-20：将分析模型与其他相关图元对齐**

```
// Pick the source analytical line to adjust to
Selection sel = app.ActiveUIDocument.Selection;
Reference refAnalytical = sel.PickObject (ObjectType.Element, "Please Pick the source analytical line to adjust to");
AnalyticalModel aModel = doc.GetElement (refAnalytical) as AnalyticalModel;
Curve aCurve = aModel.GetCurve();

// Get the reference of the start point
AnalyticalModelSelector aSelector = new AnalyticalModelSelector (aCurve);
aSelector.CurveSelector = AnalyticalCurveSelector.StartPoint;
Reference refSource = aModel.GetReference (aSelector);

// Pick the source analytical line to be adjusted
Reference refAnalytical2 = sel.PickObject (ObjectType.Element, "Please pick the source analytical line to be adjusted");
AnalyticalModel aModel2 = doc.GetElement (refAnalytical2) as AnalyticalModel;

// Get the reference of the start point
Curve aCurve2 = aModel2.GetCurve();
AnalyticalModelSelector aSelector2 = new AnalyticalModelSelector (aCurve2);
aSelector2.CurveSelector = AnalyticalCurveSelector.StartPoint;
```



```
// Can be adjusted to the middle of the line if WholeCurve is used
//aSelector2.CurveSelector = AnalyticalCurveSelector.WholeCurve;
Reference refTarget = aModel2.GetReference (aSelector2);

// Adjust the analytical line
aModel.ManuallyAdjust (refSource, refTarget, true);
```

AnalyticalModel 还提供方法来确定分析模型是否已作手动对齐, 以及将它重置回物理模型原始位置。此外, GetManualAdjustmentMatchedElements()方法对那些已作对齐的分析模型, 检索出这些图元 ID 的集合。

#### 4. 分析偏移 (Analytical Offset)

对齐分析模型的另一种方法是使用偏移。设置分析偏移不同于手动对齐分析模型。分析偏移是应用于整个分析模型并独立于任何其他图元的基本偏移。分析模型提供方法来获取和设置分析偏移, 以及确定分析偏移是否可以更改 [CanSetAnalyticalOffset()]。

#### 5. 分析模型支撑 (AnalyticalModelSupport)

AnalyticalModel 提供方法 IsElementFullySupported(), 以确定该分析模型是否是完全支撑的。为得到有关分析模型支撑的附加信息, GetAnalyticalModelSupports()方法可用来检索 AnalyticalModelSupport 对象的集合, 这些对象提供图元是如何由其他结构图元支撑的信息, 包括每个支撑 (如果有多个图元提供支撑) 的优先顺序, 提供支撑的点、曲线或面。下面的示例说明了如何在不同条件下使用 AnalyticalModelSupport 对象。

(1) 楼板和结构梁支撑信息 (Floor and Structural Beam Support Information)。在草图模式中绘制板时, 在“设计”工具栏上选择拾取支撑。如图 4-19 所示, 该板有三根支撑梁。通过对板的 AnalyticalModelSupports 集进行迭代, 可得到这三根梁和曲线支撑分析支撑类型 (CurveSupport AnalyticalSupportType)。

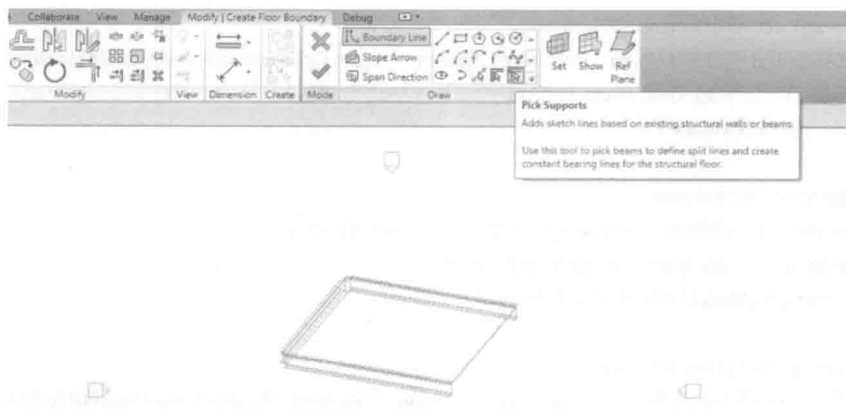


图 4-19 楼板和结构梁支撑信息

(2) 楼板和墙支撑信息 (Floor and Wall Support Information)。在选择墙作为支撑绘制楼板之后, 无法从楼板的 AnalyticalModelSupport 集合获取墙。相反, 可从墙的 AnalyticalModelSupports 集合获取楼板, 参见图 4-20。

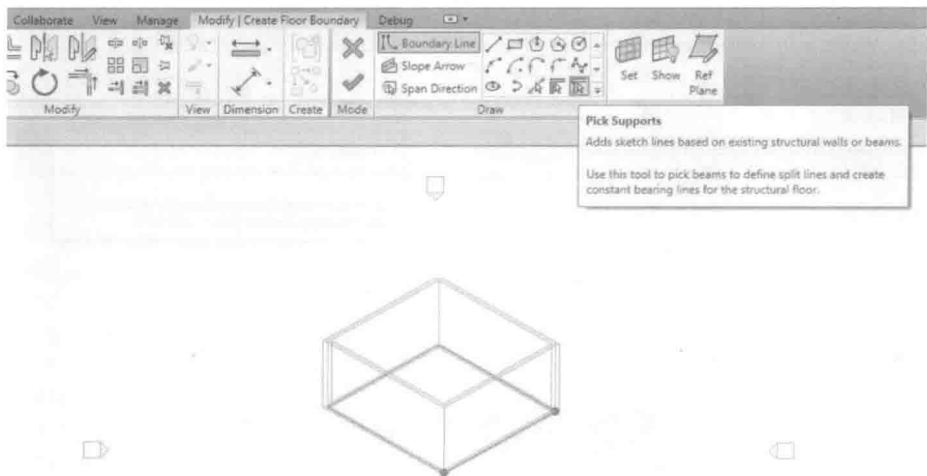


图 4-20 楼板和墙支撑信息

(3) 结构柱、梁和斜撑支撑信息 (Structural Column, Beam and Brace Support Information)。在图 4-21 中, 横梁有三个点支撑: 两根结构柱和一根结构斜撑。斜撑有三个点支撑: 两根结构柱和一根结构梁。而两根柱都无支撑图元。

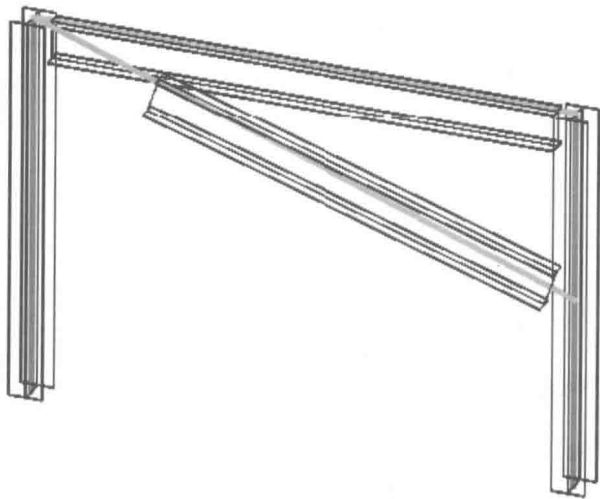


图 4-21 结构图元支撑信息

(4) 梁系统和墙支撑信息 (BeamSystem and Wall Support Information)。绘制梁系统时, 虽然可以选择墙作为支撑, 但由于 BeamSystem 无 AnalyticalModel 属性, 因此无法直接获取其支撑信息。解决方案是调用 GetBeamIds() 方法, 来检索梁的 AnalyticalModelSupport 集合, 参见图 4-22。

(5) 条形基础和墙支撑信息 (ContFooting and Wall Support Information)。对有条形基础的墙, 墙可获得条形基础的曲线支撑。用 AnalyticalModel.GetCurves() 方法获得支撑曲线。在图 4-23 中, 曲线包含两段弧线。



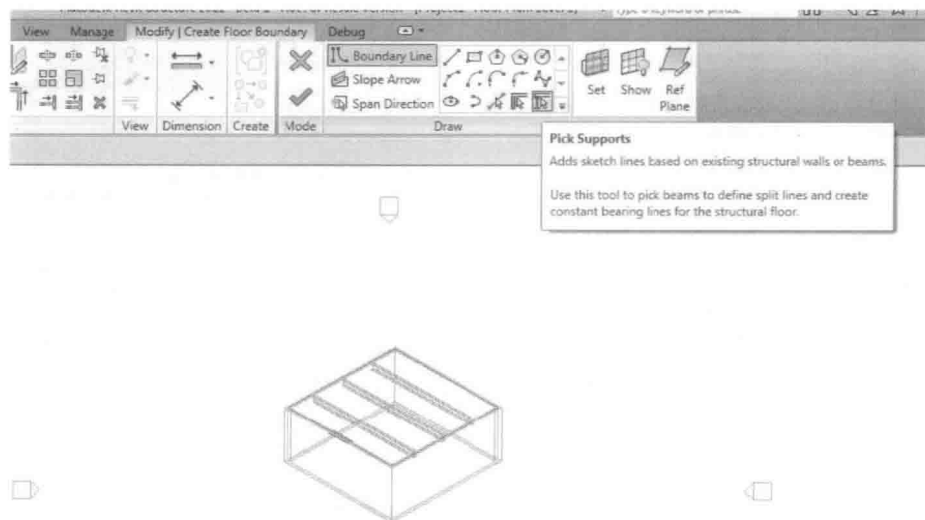


图 4-22 梁系统和墙支撑信息

(6) 独立基础和结构柱支撑信息 (Isolated Foundation and StructuralColumn Support Information)。结构柱可以有作为点支撑的独立基础。这种情况下，基础可随所支撑的结构柱移动。由 `AnalyticalModelSupport.GetSupportingElement()` 方法，可获得 `OST_Structural Foundation` 类别的族实例 `ElementId`。通常，支撑点是从 `AnalyticalModel.GetCurve()` 方法检索到的曲线的最低点。它还可以在获取独立基础族实例后，由 `GetPoint()` 方法得到的 `AnalyticalModel Point` 获取，参见图 4-24。

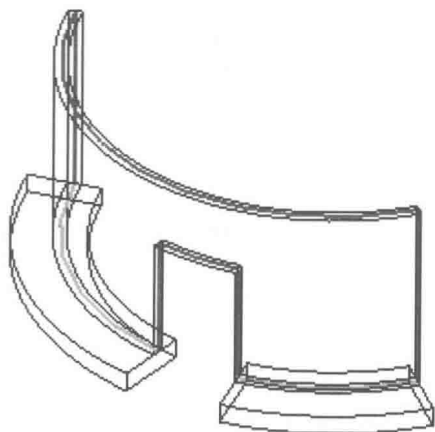


图 4-23 条形基础和墙支撑信息

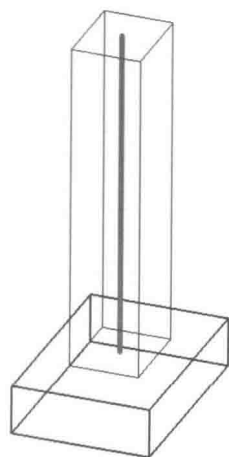


图 4-24 独立基础 (族实例) 和结构柱支撑信息

### 4.2.3 荷载 (Loads)

以下各节说明荷载设置并讨论荷载限制准则。

荷载设置 (Load Settings): API 可以访问“设置”对话框的荷载工况和载荷组合选项卡上的所有功能。

从相应的荷载工况内建参数中可以获得表 4-7 中的属性。

表 4-7 荷载工况属性和参数

属性	内建参数	属性	内建参数
工况号	LOAD_CASE_NUMBER	类别	LOAD_CASE_CATEGORY
性质	LOAD_CASE_NATURE		

LOAD\_CASE\_CATEGORY 参数返回一个 ElementId。表 4-8 说明了 Category 和 ElementId 值之间的映射。

表 4-8 荷载工况类别

荷载工况类别	内建类别	荷载工况类别	内建类别
恒载	OST_LoadCasesDead	屋顶活载	OST_LoadCasesRoofLive
活载	OST_LoadCasesLive	临时荷载	OST_LoadCasesAccidental
风荷载	OST_LoadCasesWind	温度荷载	OST_LoadCasesTemperature
雪荷载	OST_LoadCasesSnow	地震荷载	OST_LoadCasesSeismic

以下 Document 创建方法可创建相应的子类：

- NewLoadUsage() 创建荷载用法。
- NewLoadNature() 创建荷载性质。
- NewLoadCase() 创建荷载工况。
- NewLoadCombination() 创建荷载组合，参见代码 4-21。
- NewPointLoad() 创建点荷载（重载允许指定荷载主体图元作为参照）。可以选择性地指定荷载类型和草图平面。
- NewLineLoad() 创建线荷载（重载允许指定主体图元作为参照或图元）。可以选择性地指定荷载类型和草图平面。
- NewAreaLoad() 创建面荷载（重载允许指定主体图元作为参照或图元）。可以选择性地指定荷载类型和草图平面。

由于它们都是 Element 的子类，因此，可以使用 Document.Delete()删除它们。

代码 4-21: NewLoadCombination()

```
public LoadCombination NewLoadCombination (string name,
    int typeInd, int stateInd, double[] factors, LoadCaseArray cases, LoadCombinationArray combinations, LoadUsageArray usages);
```

对于 NewLoadCombination()方法，因子数量必须不小于工况和组合之和。例如，如果 cases.Size 为 M，combinations.Size 为 N，Factors.Size 应不小于 M+N。第一个因子 M 按顺序映射到 M 个工况，其后的因子 N 映射到 N 种组合。

注意：荷载组合不包括其自身。

LoadCase 和 LoadNature 类中没有 Duplicate()方法。要实现此功能，首先必须使用 NewLoadCase() [或 NewLoadNature()] 方法创建一个新的 LoadCase（或 LoadNature）对



象, 然后从现有 LoadCase (或 LoadNature) 复制相应的属性和参数。

代码 4-22 是一段用 VB.NET 创建常驻点荷载的最短演示代码。

代码 4-22: NewPointLoad()

```
" NewPointLoad with a host (in VB.NET)
"Select a beam, and get a curve of its analytical model.

Dim ref As Reference = rvtUIDoc.Selection.PickObject ( _
    ObjectType.Element, "Select a beam")
Dim elem As Element = ref.Element
Dim aModel As AnalyticalModel = elem.GetAnalyticalModel
Dim aCurve As Curve = aModel.GetCurve

" For simplicity, we assume we want to add a point load
" at the start point of the curve.

Dim aSelector As New AnalyticalModelSelector (aCurve)
aSelector.CurveSelector = AnalyticalCurveSelector.StartPoint
Dim startPointRef As Reference = aModel.GetReference (aSelector)

" Create a hosted point load

Dim force As XYZ = New XYZ (0.0, 0.0, -10000.0)
Dim moment As XYZ = New XYZ (-100.0, 100.0, 100.0)

Dim myPointLoad As PointLoad = _
    rvtDoc.Create.NewPointLoad ( _
        startPointRef, force, moment, False, Nothing, Nothing)
```

#### 4.2.4 分析链接 (Analysis Link)

在 Revit Structure 中, 分析模型会随着创建物理模型自动生成。分析模型被链接到结构分析应用程序, 且物理模型通过 Revit Structure API 根据结果作自动更新。一些第三方软件开发商已经为它们的结构分析应用程序提供了双向链接。包括:

- ADAPT-Builder Suite from ADAPT Corporation ([www.adaptsoft.com/revitstructure/](http://www.adaptsoft.com/revitstructure/))。
- Fastrak and S-Frame from CSC ([www.cscworld.com](http://www.cscworld.com/))。
- ETABS from CSI ([www.csiberkeley.com](http://www.csiberkeley.com/))。
- RFEM from Dlubal ([www.dlubal.com/FEA-Software-RFEM-integrates-with-Revit-Structure.aspx](http://www.dlubal.com/FEA-Software-RFEM-integrates-with-Revit-Structure.aspx))。
- Advance Design, VisualDesign, Arche, Effel and SuperSTRESS from GRAITEC ([www.graitec.com/En/revit.asp](http://www.graitec.com/En/revit.asp))。
- Scia Engineer from Nemetschek ([www.scia-online.com/en/revit.html](http://www.scia-online.com/en/revit.html))。
- GSA from Oasys Software (Arup) ([www.oasys-software.com/products](http://www.oasys-software.com/products))。
- ProDESK from Prokon Software Consultants ([www.prokon.com](http://www.prokon.com/))。
- RAM Structural System from Bentley ([www.bentley.com/en-US/Products/RAM+Structural+System/](http://www.bentley.com/en-US/Products/RAM+Structural+System/))。

- RISA-3D and RISAFloor from RISA Technologies ([www.risatech.com/partner/revit\\_structure.asp](http://www.risatech.com/partner/revit_structure.asp))。
- SOFiSTiK Structural Desktop Suite from SOFiSTiK (<http://www.sofistik.com/revit-to-sofistik>)。
- SPACE GASS from SPACE GASS ([www.spacegass.com/index.asp?resend=/revit.asp](http://www.spacegass.com/index.asp?resend=/revit.asp))。
- Revit Structure STAAD.Pro interface from Structural Integrators ([structuralintegrators.com/products/si\\_xchange.php](http://structuralintegrators.com/products/si_xchange.php))。

将 Revit Structure 链接到其他分析应用程序的关键，是建立不同对象模型中对象之间的映射关系。这意味着整合的难度和程度取决于两个对象模型之间的相似性。

例如，在产品的设计过程中，设计一张表，表中至少包含表 4-9 所列对象映射的前两列：一列用于 Revit Structure API，另一列用于其他结构分析应用程序。

表 4-9 Revit 和分析应用程序对象映射

Revit Structural API	分析应用程序	导入 Revit
StructuralColumn	Column	NewStructuralColumn
Property:		
...		
Location		Read-only;
Parameter:		
...		
Analyze as		Editable;
AnalyticalModel:		
...		
Profile		Read-only;
RigidLink		Read-only;
...		
Material:		
...		

4.2.5 分析连接 (Analytical Links)

分析连接是一个连接两个独立分析节点的图元，具有诸如固定状态等属性。分析连接可通过 Revit 建模期间根据某些规则从分析梁到分析柱自动创建。它们也可以在 Revit 用户界面中或使用 Revit API 手动创建。

Revit API 中，分析连接由 AnalyticalLink 类来表示。Fixity 值可由关联的 AnalyticalLinkType 获得。

代码 4-23 演示了如何读取文件中所有的分析连接，并在对话框中显示自动生成和手动创建的分析连接的数量。

**代码 4-23: 读取分析连接**

```
public void ReadAnalyticalLinks ( Document document )
{
    FilteredElementCollector collectorAnalyticalLinks = new FilteredElementCollector ( document );
    collectorAnalyticalLinks.OfClass ( typeof ( AnalyticalLink ));

    IEnumerable<AnalyticalLink> alinks = collectorAnalyticalLinks.ToElements().Cast<AnalyticalLink>();
    int nAutoGeneratedLinks = 0;
    int nManualLinks = 0;
    foreach ( AnalyticalLink alink in alinks )
    {
        if ( alink.IsAutoGenerated() == true )
            nAutoGeneratedLinks++;
        else
            nManualLinks++;
    }
    string msg = "Auto-generated AnalyticalLinks: " + nAutoGeneratedLinks;
    msg += "\nManually created AnalyticalLinks: " + nManualLinks;
    TaskDialog.Show ( "AnalyticalLinks", msg );
}
```

静态方法 `AnalyticalLink.Create()` 创建一个新的分析连接, 而不是直接连接两个图元, 连接在两个“中心”之间创建。`Hub` 类表示两个或更多 Autodesk Revit 图元之间的连接。

代码 4-24 在两个选定的 `FamilyInstance` 对象之间创建了一个新的分析连接。它使用过滤器来查找模型中所有的“中心”, 然后用 `GetHub()` 方法搜索所有“中心”, 以找出参照各自族实例分析模型 ID 的“中心”连接。

**代码 4-24: 创建一个新的分析连接**

```
public void CreateLink ( Document doc, FamilyInstance fi1, FamilyInstance fi2 )
{
    FilteredElementCollector hubCollector = new FilteredElementCollector ( doc );
    hubCollector.OfClass ( typeof ( Hub )); //Get all hubs
    ICollection<Element> allHubs = hubCollector.ToElements();
    FilteredElementCollector linktypeCollector = new FilteredElementCollector ( doc );
    linktypeCollector.OfClass ( typeof ( AnalyticalLinkType ));
    ElementId firstLinkType = linktypeCollector.ToElementIds().First(); //Get the first analytical link type.

    // Get hub Ids from two selected family instance items
    ElementId startHubId = GetHub ( fi1.GetAnalyticalModel().Id, allHubs );
    ElementId endHubId = GetHub ( fi2.GetAnalyticalModel().Id, allHubs );

    Transaction tran = new Transaction ( doc, "Create Link" );
    tran.Start();
    //Create a link between these two hubs.
    AnalyticalLink createdLink = AnalyticalLink.Create ( doc, firstLinkType, startHubId, endHubId );
    tran.Commit();
}
```



```
//Get the first Hub on a given AnalyticalModel element
private ElementId GetHub (ElementId hostId, ICollection<Element> allHubs)
{
    foreach (Element ehub in allHubs)
    {
        Hub hub = ehub as Hub;
        ConnectorManager manager = hub.GetHubConnectorManager( );
        ConnectorSet connectors = manager.Connectors;
        foreach (Connector connector in connectors)
        {
            ConnectorSet refConnectors = connector.AllRefs;
            foreach (Connector refConnector in refConnectors)
            {
                if (refConnector.Owner.Id == hostId)
                {
                    return hub.Id;
                }
            }
        }
    }
    return ElementId.InvalidElementId;
}
```

## 4.3 Revit MEP

Revit API 的 Revit MEP 部分，提供对 Revit 模型中暖通空调（HVAC）和管道数据的读取和写入访问。包括：

- 遍历系统中的风管、管道、管件、连接件。
- 添加、删除和更改风管、管道及其他设备。
- 获取和设置系统属性。
- 确定系统是否连接良好。
- 访问机械设置。
- 管理布线配置。

### 4.3.1 MEP 图元创建（MEP Element Creation）

风管和管道系统的相关图元可以使用 Autodesk.Revit. Creation. Document 类中的以下方法创建：

- NewDuct。
- NewFlexDuct。
- NewPipe。
- NewFlexPipe。
- NewMechanicalSystem。
- NewPipingSystem。



- NewCrossFitting。
- NewElbowFitting。
- NewTakeoffFitting。
- NewTeeFitting。
- NewTransitionFitting。
- NewUnionFitting。

#### 1. 创建管道和风管 ( Create Pipes and Ducts )

有三种方法来创建新的风管、软风管、管道和软管。它们可以在两个点之间、两个连接件之间，或者一个点和一个连接件之间创建。此外，在两个点之间创建这些 MEPCurves 类型之一时，可以使用其对应类的静态方法 Create()。

代码 4-25 使用 Autodesk.Revit. Creation. Document. NewPipe() 方法，在两点之间创建新管道。新的软管、风管和软风管也都可以类似地创建。

**代码 4-25：用 NewPipe() 方法新建管道**

```
public Pipe CreateNewPipe (Document document)
{
    // find a pipe type

    FilteredElementCollector collector = new FilteredElementCollector (document);
    collector.OfClass (typeof (PipeType));
    PipeType pipeType = collector.FirstElement() as PipeType;

    Pipe pipe = null;
    if (null != pipeType)
    {
        // create pipe between 2 points
        XYZ p1 = new XYZ (0, 0, 0);
        XYZ p2 = new XYZ (10, 0, 0);

        pipe = document.Create.NewPipe (p1, p2, pipeType);
    }

    return pipe;
}
```

代码 4-26 演示了如何使用静态方法 FlexPipe.Create() 创建软管。类似地，风管、软风管也都可以两点之间创建。

**代码 4-26：用静态方法 Create() 新建软管**

```
public FlexPipe CreateFlexPipe (Document document, Level level)
{
    // find a pipe type
    FilteredElementCollector collector = new FilteredElementCollector (document);
    collector.OfClass (typeof (FlexPipeType));
    ElementId pipeTypeId = collector.FirstElementId();
}
```



```
// find a pipe system type
FilteredElementCollector sysCollector = new FilteredElementCollector (document);
sysCollector.OfClass (typeof (PipingSystemType));
ElementId pipeSysTypeId = sysCollector.FirstElementId();

FlexPipe pipe = null;
if (pipeTypeId != ElementId.InvalidElementId && pipeSysTypeId != ElementId.InvalidElementId)
{
    // create flex pipe with 3 points
    List<XYZ> points = new List<XYZ>();
    points.Add (new XYZ (0, 0, 0));
    points.Add (new XYZ (10, 10, 0));
    points.Add (new XYZ (10, 0, 0));

    pipe = FlexPipe.Create (document, pipeSysTypeId, pipeTypeId, level.Id, points);
}

return pipe;
}
```

在创建管道之后, 如果希望更改管道直径, 请获取 RBS\_PIPE\_DIAMETER\_PARAM 内建参数。管道的 Diameter 属性是只读的, 参见代码 4-27。

#### 代码 4-27: 更改管道直径

```
public void ChangePipeSize (Pipe pipe)
{
    Parameter parameter = pipe.get_Parameter (BuiltInParameter.RBS_PIPE_DIAMETER_PARAM);

    string message = "Pipe diameter: " + parameter.AsValueString();

    parameter.Set (0.5); // set to 6" (0.5ft)

    message += "\nPipe diameter after set: " + parameter.AsValueString();

    MessageBox.Show (message, "Revit");
}
```

新建风管或管道的另一种常见方法, 是在两个现有连接件之间创建, 见代码 4-28。在这个例子中, 假设 Revit MEP 中两个连接件图元已选定, 一个是机械设备, 另一个是与设备上的供气连接件排成一线的、带有连接件的风管管件。

#### 代码 4-28: 在两个连接件之间添加一段风管

```
public Duct CreateDuctBetweenConnectors (UIDocument uiDocument)
{
    // prior to running this example
    // select some mechanical equipment with a supply air connector
    // and an elbow duct fitting with a connector in line with that connector
    Connector connector1 = null, connector2 = null;
```





```
ConnectorSetIterator csi = null;
ElementSet selection = uiDocument.Selection.Elements;
// First find the selected equipment and get the correct connector
foreach (Element e in selection)
{
    if (e is FamilyInstance)
    {
        FamilyInstance fi = e as FamilyInstance;
        Family family = fi.Symbol.Family;
        if (family.FamilyCategory.Name == "Mechanical Equipment")
        {
            csi = fi.MEPModel.ConnectorManager.Connectors.ForwardIterator();
            while (csi.MoveNext())
            {
                Connector conn = csi.Current as Connector;
                if (conn.Direction == FlowDirectionType.Out &&
                    conn.DuctSystemType == DuctSystemType.SupplyAir)
                {
                    connector1 = conn;
                    break;
                }
            }
        }
    }
}
// next find the second selected item to connect to
foreach (Element e in selection)
{
    if (e is FamilyInstance)
    {
        FamilyInstance fi = e as FamilyInstance;
        Family family = fi.Symbol.Family;
        if (family.FamilyCategory.Name != "Mechanical Equipment")
        {
            csi = fi.MEPModel.ConnectorManager.Connectors.ForwardIterator();
            while (csi.MoveNext())
            {
                if (null == connector2)
                {
                    Connector conn = csi.Current as Connector;

                    // make sure to choose the connector in line with the first connector
                    if (Math.Abs (conn.Origin.Y - connector1.Origin.Y) < 0.001)
                    {
                        connector2 = conn;
                        break;
                    }
                }
            }
        }
    }
}
```



```

    }
}

Duct duct = null;
if (null != connector1 && null != connector2)
{
    // find a duct type
    FilteredElementCollector collector =
        new FilteredElementCollector (uiDocument.Document);
    collector.OfClass (typeof (DuctType));

    // Use Linq query to make sure it is one of the rectangular duct types
    var query = from element in collector
                where element.Name.Contains ("Mitered Elbows") == true
                select element;

    // use extension methods to get first duct type
    DuctType ductType = collector.Cast<DuctType>( ).First<DuctType>( );

    if (null != ductType)
    {
        duct = uiDocument.Document.Create.NewDuct (connector1, connector2, ductType);
    }
}

return duct;
}

```

图 4-25 是在选择并联风机动力变风量装置 (VAV Unit) 和矩形肘管风管管件之后, 代码 4-28 运行的结果。

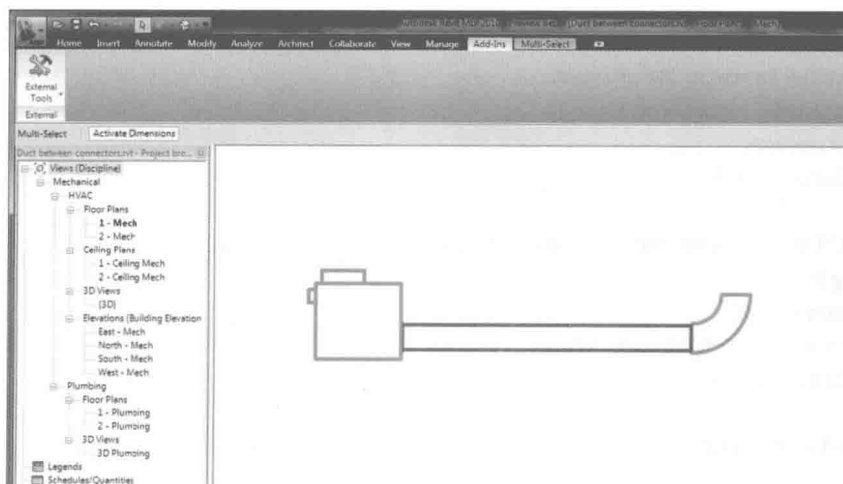


图 4-25 选定图元之间新添加的风管



## 2. 内衬和隔热层 (Lining and Insulation)

管道或风管隔热层和内衬可以作为与风管或管道相关的单独对象添加。管道或风管相关的隔热层图元 ID 可以使用静态方法 `InsulationLiningBase.GetInsulationIds()` 来检索, 而内衬图元 ID 可以使用静态方法 `InsulationLiningBase.GetLiningIds()` 来检索。

新建与给定风管、管道、管件、附件或连接件相关的隔热层, 使用相应的静态方法: `DuctInsulation.Create()` 或 `PipeInsulation.Create()`。`DuctLining.Create()` 可用于创建内衬的新实例, 应用于给定风管、管件或附件的内部。

## 3. 占位符风管和管道 (Placeholder Ducts and Pipes)

Revit API 提供了占位符图元的能力, 在布局的确切设计尚未知的情况下, 将占位符图元放入系统中。使用占位符风管和管道可以给予设计还是未知的系统一个连接良好的系统布局, 然后可以在后阶段最终设计中详细制作。

两个静态方法 `Duct.CreatePlaceholder()` 和 `Pipe.CreatePlaceholder()` 都可创建占位符图元。风管和管道的 `IsPlaceholder` 属性指示它们是否是个占位符图元。

当准备创建占位符的实际风管和管道时, 使用 `MechanicalUtils.ConvertDuctPlaceholders()` 和 `PlumbingUtils.ConvertPipePlaceholders()` 方法将一组占位符图元转换为风管和管道。一旦转换成功, 占位符图元即被删除, 而新风管、管道和管件图元则被创建并建立连接。

## 4. 新建系统 (Creating a New System)

可以使用 Revit API 创建新的管道和机械系统。`NewPipingSystem()` 和 `NewMechanicalSystem()` 都采用 `Connector`, 它是基础设备连接件, 如管道系统的热水器, 或机械系统的风机。它们还采用将要添加到系统的连接器的 `ConnectorSet`, 如管道系统中洗涤池上的水龙头。创建新系统所需的最后一点信息是 `NewPipingSystem()` 的 `PipeSystemType` 或是 `NewMechanicalSystem()` 的 `DuctSystemType`。

代码 4-29 从一件选定的机械设备 (如风机) 和所有选定的风道末端, 新建供气风管系统。

### 代码 4-29: 新建机械系统

```
// create a connector set for new mechanical system
ConnectorSet connectorSet = new ConnectorSet();
// Base equipment connector
Connector baseConnector = null;

// Select a Parallel Fan Powered VAV and some Supply Diffusers
// prior to running this example
ConnectorSetIterator csi = null;
ElementSet selection = document.Selection.Elements;
foreach (Element e in selection)
{
    if (e is FamilyInstance)
    {
        FamilyInstance fi = e as FamilyInstance;
```



```

Family family = fi.Symbol.Family;
// Assume the selected Mechanical Equipment is the base equipment for new system
if (family.FamilyCategory.Name == "Mechanical Equipment")
{
    //Find the "Out" and "SupplyAir" connector on the base equipment
    if (null != fi.MEPModel)
    {
        csi = fi.MEPModel.ConnectorManager.Connectors.ForwardIterator();
        while (csi.MoveNext())
        {
            Connector conn = csi.Current as Connector;
            if (conn.Direction == FlowDirectionType.Out &&
                conn.DuctSystemType == DuctSystemType.SupplyAir)
            {
                baseConnector = conn;
                break;
            }
        }
    }
}
else if (family.FamilyCategory.Name == "Air Terminals")
{
    // add selected Air Terminals to connector set for new mechanical system
    csi = fi.MEPModel.ConnectorManager.Connectors.ForwardIterator();
    csi.MoveNext();
    connectorSet.Insert (csi.Current as Connector);
}
}
}
MechanicalSystem mechanicalSys = null;
if (null != baseConnector && connectorSet.Size > 0)
{
    // create a new SupplyAir mechanical system
    mechanicalSys = document.Create.NewMechanicalSystem (baseConnector, connectorSet, DuctSystemType.SupplyAir);
}
}

```

#### 4.3.2 连接件 (Connectors)

如上一节所述，新风管和管道可以在两个连接件之间创建。连接件用以连接主体与风管、管道及电气设备，它们由连接件的 Domain 属性获得。连接件存在于机械设备和风管及管道中。

遍历系统，通过检查 IsConnected 属性和 AllRefs 属性，可检查系统基础设备上的连接件，确定连接件所连接的是什么。查找物理连接时，重要的是要检查连接件的 ConnectionType。Revit 既有物理连接件又有逻辑连接件，但应用程序中仅物理连接件是可见的。图 4-26 显示了两种类型的物理连接件，即端部连接件和曲线连接件。

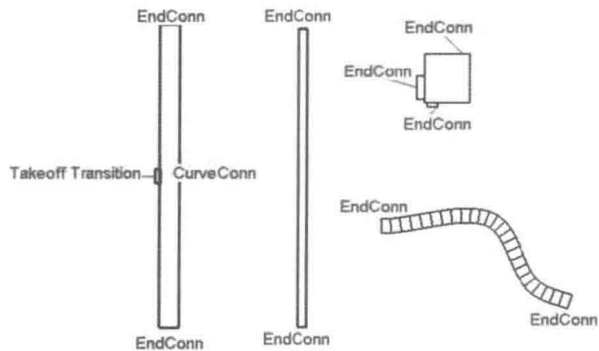


图 4-26 物理连接件

代码 4-30 演示了如何确定连接件的拥有者，若存在任何连接，则连同确定连接类型。

#### 代码 4-30：确定连接件附着的是什么

```
public void GetElementAtConnector (Connector connector)
{
    MEPSsystem mepSystem = connector.MEPSsystem;
    if (null != mepSystem)
    {
        string message = "Connector is owned by: " + connector.Owner.Name;

        if (connector.IsConnected == true)
        {
            ConnectorSet connectorSet = connector.AllRefs;
            ConnectorSetIterator csi = connectorSet.ForwardIterator();
            while (csi.MoveNext())
            {
                Connector connected = csi.Current as Connector;
                if (null != connected)
                {
                    // look for physical connections
                    if (connected.ConnectorType == ConnectorType.EndConn ||
                        connected.ConnectorType == ConnectorType.CurveConn ||
                        connected.ConnectorType == ConnectorType.PhysicalConn)
                    {
                        message += "\nConnector is connected to: " + connected.Owner.Name;
                        message += "\nConnection type is: " + connected.ConnectorType;
                    }
                }
            }
        }
        else
        {
            message += "\nConnector is not connected to anything.";
        }
    }
}
```



```

        MessageBox.Show (message, "Revit");
    }
}

```

图 4-27 的对话框是从一件机械设备的连接件上运行代码 4-30 的结果。

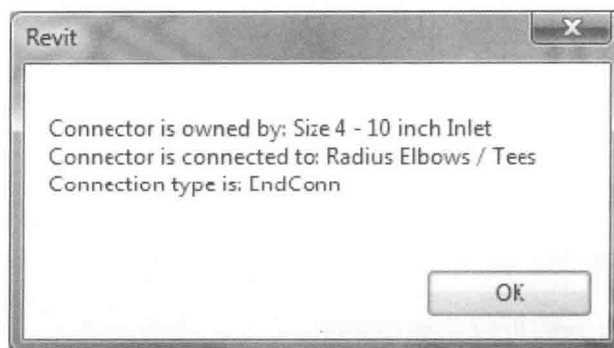


图 4-27 连接件信息

### 4.3.3 族创建 (Family Creation)

在 Revit 族文件中创建机械设备时，需要添加连接件以允许将设备连接到系统。使用参考平面来放置连接件和设定连接件的系统类型后，风管、电气和管道连接件都可以类似地添加。

由 ConnectorElement 类提供的静态方法重载：

- CreateCableTrayConnector。
- CreateConduitConnector。
- CreateDuctConnector。
- CreateElectricalConnector。
- CreatePipeConnector。

以上每种方法都有个需要外加 Edge 参数的第二个重载，允许连接件图元以给定面的内环为中心创建。代码 4-31 演示了如何将两个管道连接件添加到拉伸图元的面上，并对其设置一些属性。

#### 代码 4-31：添加管道连接件

```

public void CreatePipeConnectors (UIDocument uiDocument, Extrusion extrusion)
{
    // get the faces of the extrusion
    Options geoOptions = uiDocument.Document.Application.Create.NewGeometryOptions( );
    geoOptions.View = uiDocument.Document.ActiveView;
    geoOptions.ComputeReferences = true;

    List<PlanarFace> planarFaces = new List<PlanarFace>( );
    Autodesk.Revit.DB.GeometryElement geoElement = extrusion.get_Geometry (geoOptions);
    foreach (GeometryObject geoObject in geoElement)
    {

```



```
Solid geoSolid = geoObject as Solid;
if (null != geoSolid)
{
    foreach (Face geoFace in geoSolid.Faces)
    {
        if (geoFace is PlanarFace)
        {
            planarFaces.Add (geoFace as PlanarFace);
        }
    }
}

if (planarFaces.Count > 1)
{
    // Create the Supply Hydronic pipe connector
    ConnectorElement connSupply = ConnectorElement.CreatePipeConnector (uiDocument.Document,
PipeSystemType.SupplyHydronic,
                                                                    planarFaces[0].Reference);

    Parameter param = connSupply.get_Parameter (BuiltInParameter.CONNECTOR_RADIUS);
    param.Set (1.0); // 1' radius
    param = connSupply.get_Parameter (BuiltInParameter.RBS_PIPE_FLOW_DIRECTION_PARAM);
    param.Set (2);

    //Create the Return Hydronic pipe connector
    ConnectorElement connReturn = ConnectorElement.CreatePipeConnector (uiDocument.Document,
PipeSystemType.ReturnHydronic,
                                                                    planarFaces
[1].Reference);
    param = connReturn.get_Parameter (BuiltInParameter.CONNECTOR_RADIUS);
    param.Set (0.5); // 6" radius
    param = connReturn.get_Parameter (BuiltInParameter.RBS_PIPE_FLOW_DIRECTION_PARAM);
    param.Set (1);
}
}
```

用机械设备样板并传入一个 2'×2'×1' 的拉伸件新建一个族文件, 在该族文件中运行代码 4-31, 图 4-28 示出了运行结果。请注意, 连接件放置在平面的中心。

#### 4.3.4 机械设置 (Mechanical Settings)

在“MEP 设置”下面的“管理”选项卡上有许多可用设置——“机械设置”, 也可通过 Revit API 使用这些设置。

##### 1. 管道设置 (Pipe Settings)

PipeSettings 类提供访问图 4-29 所示的设置, 如管道尺寸后缀和管道连接件公差。每



个文件都有一个 PipeSettings 对象, 并可通过静态方法 PipeSettings.GetPipeSettings() 来访问。

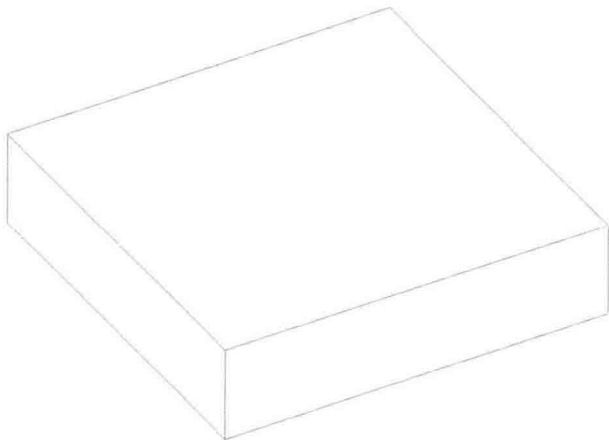


图 4-28 在拉伸件上创建两个连接件

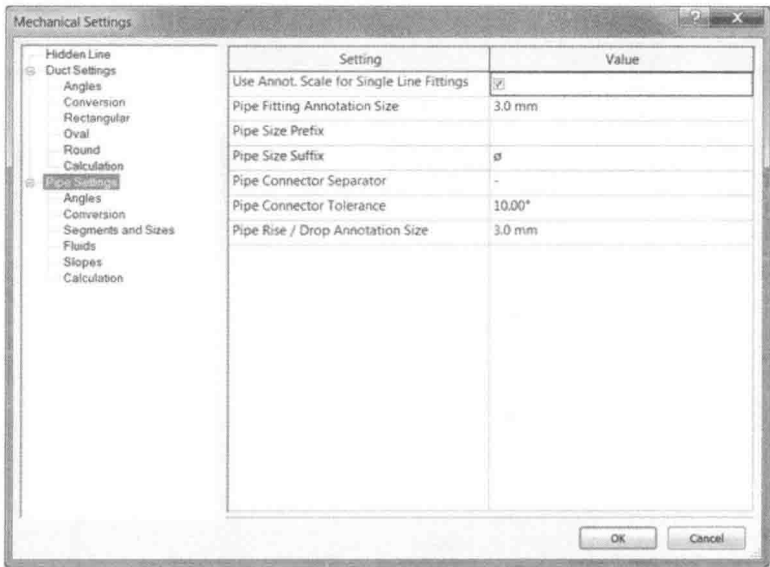


图 4-29 管道设置

2. 管件角度 (Fitting Angles)

管道的管件角度使用设置可从 PipeSettings 类的下列属性和方法获得, 参见图 4-30。

- PipeSettings.FittingAngleUsage()。
- PipeSettings.GetSpecificFittingAngles()。
- PipeSettings.GetSpecificFittingAngleStatus()。
- PipeSettings.SetSpecificFittingAngleStatus()。

3. 管段和尺寸 (Segments and Sizes)

用户界面中“管道设置”下的设置——管段和尺寸在 API 中也同样可用, 参见图 4-31。



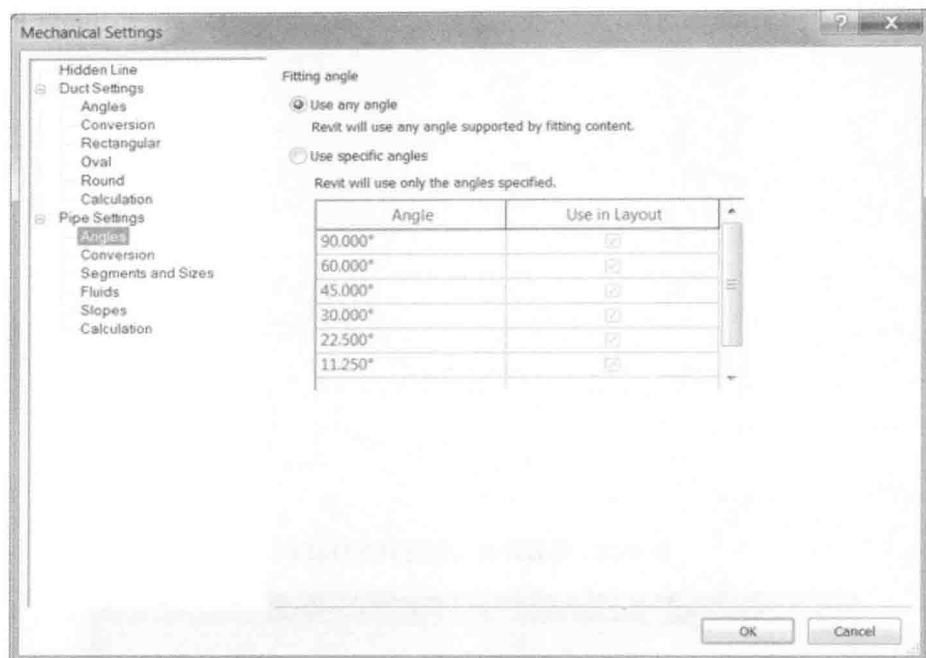


图 4-30 管道的管件角度

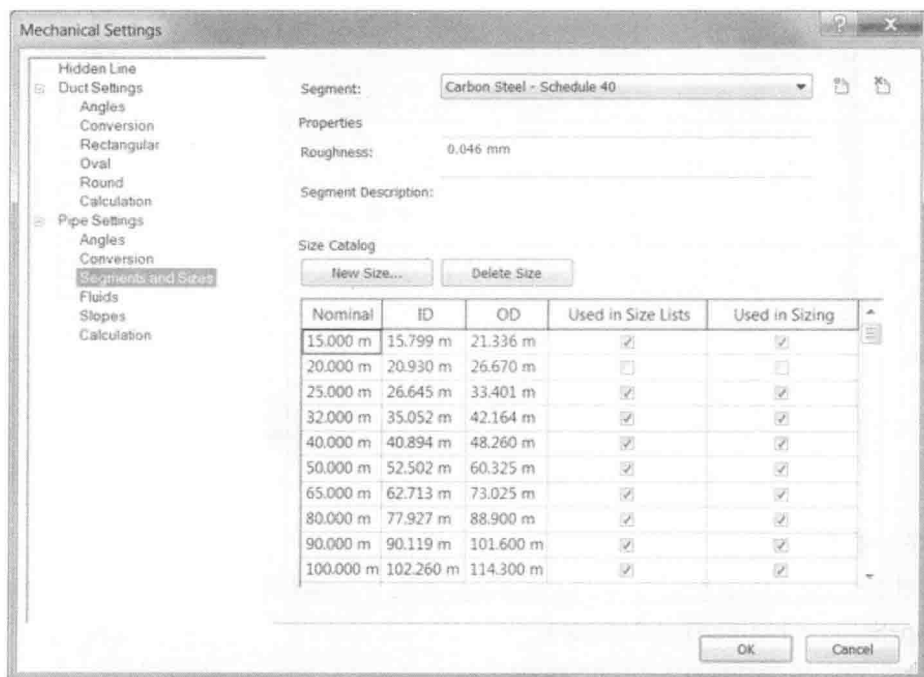


图 4-31 管段和尺寸

此信息可通过 Segment 和 MEPSize 类获得。Segment 类表示一段 MEPCurve，它包含某种材料和一组可用尺寸集。MEPSize 类表示管道尺寸。使用过滤器可找出可用的 Segments。代码 4-32 演示了如何获取图 4-31 对话框中的一些信息，输出结果见图 4-32。



### 代码 4-32: 在管道设置中遍历管道尺寸

```
FilteredElementCollector collectorPipeType = new FilteredElementCollector (document);
collectorPipeType.OfClass (typeof (Segment));

IEnumerable<Segment> segments = collectorPipeType.ToElements().Cast<Segment>();
foreach (Segment segment in segments)
{
    StringBuilder strPipeInfo = new StringBuilder ();
    strPipeInfo.AppendLine ("Segment: " + segment.Name);

    strPipeInfo.AppendLine ("Roughness: " + segment.Roughness);

    strPipeInfo.AppendLine ("Pipe Sizes: ");
    double dLengthFac = 304.8;    // used to convert stored units from ft to mm for display
    foreach (MEPSize size in segment.GetSizes())
    {
        strPipeInfo.AppendLine (string.Format ("Nominal: {0: F3}, ID: {1: F3}, OD: {2: F3}",
                                                size.NominalDiameter * dLengthFac , size.InnerDiameter * dLengthFac ,
                                                size.OuterDiameter * dLengthFac));
    }

    TaskDialog.Show ("PipeSetting Data", strPipeInfo.ToString());
    break;
}
```

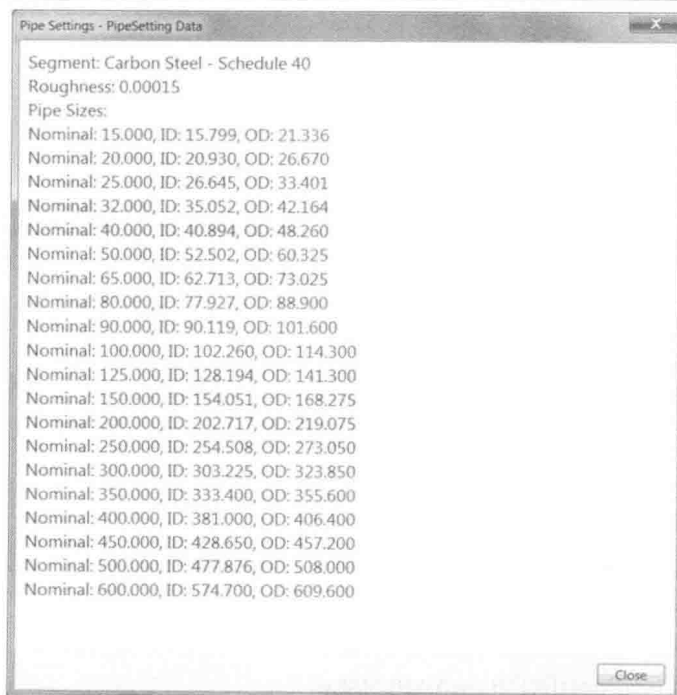


图 4-32 代码 4-32 的输出结果



要向列表中添加新尺寸, 请使用 `Segment.AddSize()` 方法。使用 `Segment.RemoveSize()` 可删除某个公称直径尺寸。

#### 4. 坡度 (Slopes)

`PipeSettings` 类还提供了对用户界面中“管道设置”下的坡度值的访问——`Slopes`。使用 `GetPipeSlopes()` 检索坡度值列表。`PipeSettings.SetPipeSlopes()` 提供了一次设置全部坡度值的功能, 而 `PipeSettings.AddPipeSlope()` 添加单个管道的坡度。Revit 坡度值存储为百分比 (0~100), 参见图 4-33。

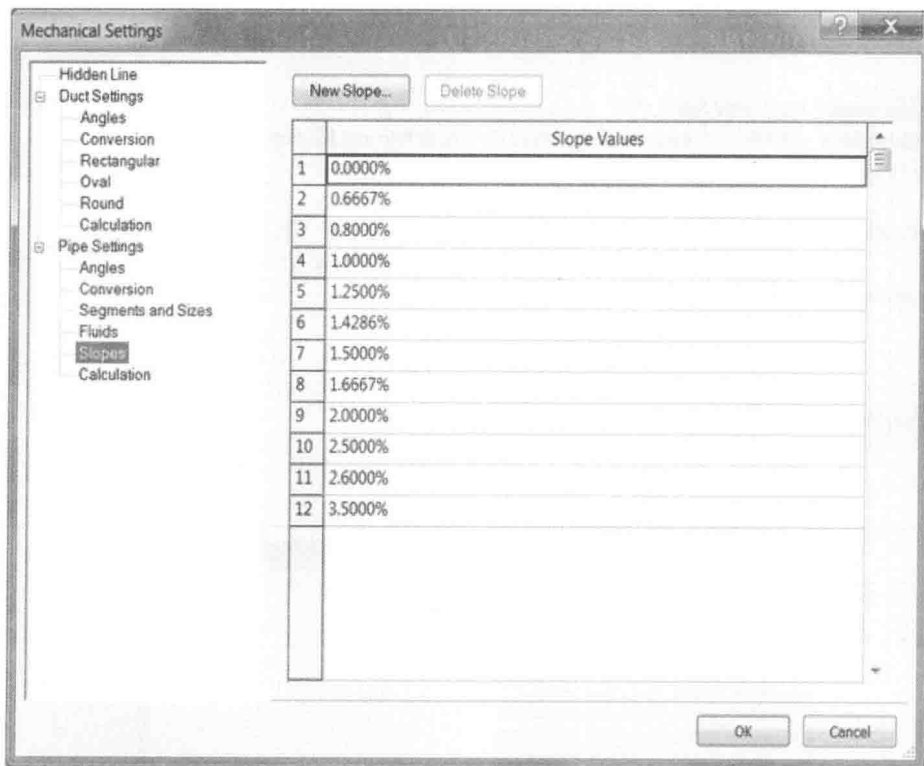


图 4-33 管道坡度值

#### 5. 风管设置 (Duct Settings)

`DuctSettings` 类支持访问图 4-34 所示的设置, 如风管管件注释尺寸和空气密度。每个文件都有一个 `DuctSettings` 对象, 它可以通过静态方法 `DuctSettings.GetDuctSettings()` 来访问。

#### 6. 风管管件角度 (Duct Fitting Angles)

风管管件角度用法设置可从 `DuctSettings` 类的下列属性和方法获得, 参见图 4-35。

- `DuctSettings.FittingAngleUsage()`。
- `DuctSettings.GetSpecificFittingAngles()`。
- `DuctSettings.GetSpecificFittingAngleStatus()`。
- `DuctSettings.SetSpecificFittingAngleStatus()`。

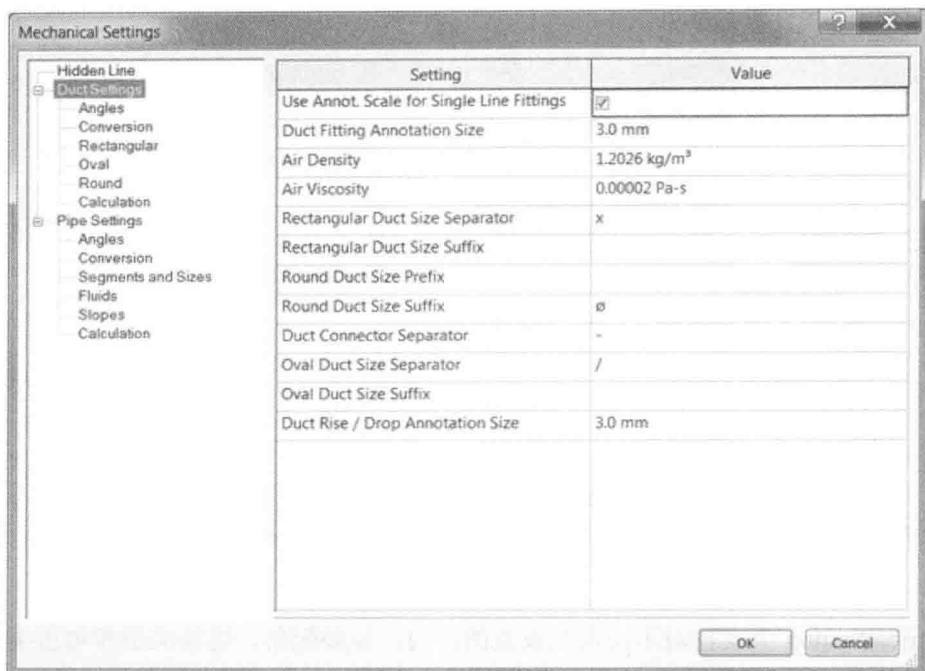


图 4-34 风管设置

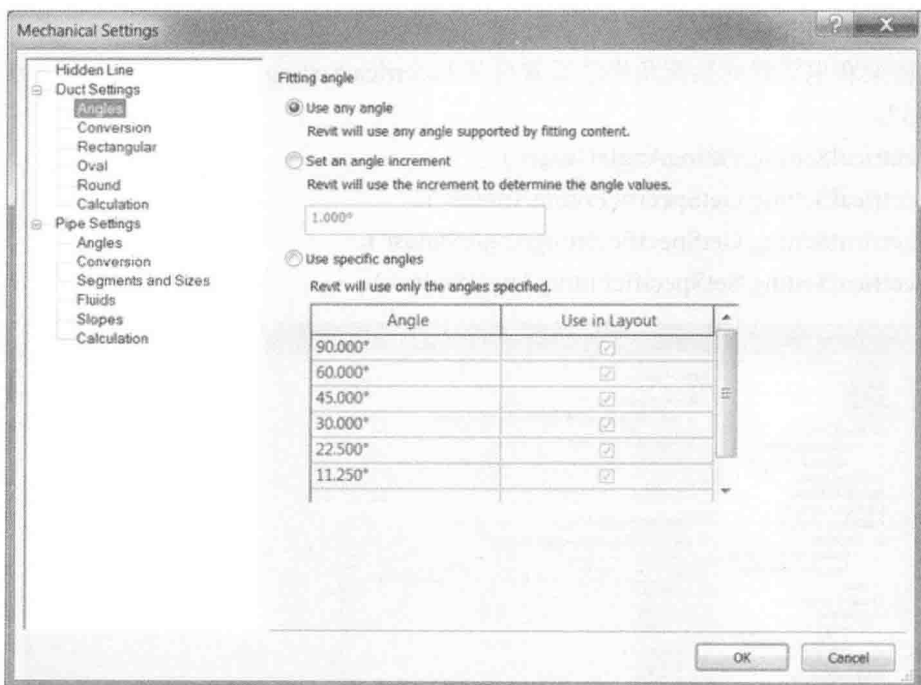


图 4-35 风管管件角度

### 4.3.5 电气设置 (Electrical Settings)

在“MEP 设置”下面“管理”选项卡上有一些可用设置——“电气设置”，也可通过



Revit API 使用这些设置，参见图 4-36。

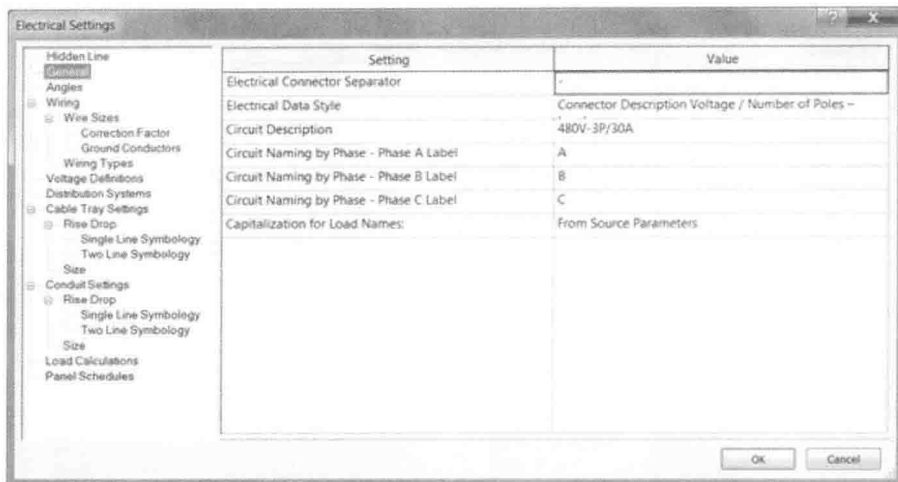


图 4-36 电气设置

ElectricalSetting 类支持对不同电气设置的访问，如装配角、缆线类型和电压类型。每个文件都有一个 ElectricalSetting 对象，并可通过静态方法 ElectricalSetting.GetElectricalSettings() 访问。

### 1. 装配角 (Fitting Angles)

电缆桥架和电缆管的装配角用法设置可从 ElectricalSetting 类的下列属性和方法获得，参见图 4-37。

- ElectricalSetting.FittingAngleUsage()。
- ElectricalSetting.GetSpecificFittingAngles()。
- ElectricalSetting.GetSpecificFittingAngleStatus()。
- ElectricalSetting.SetSpecificFittingAngleStatus()。

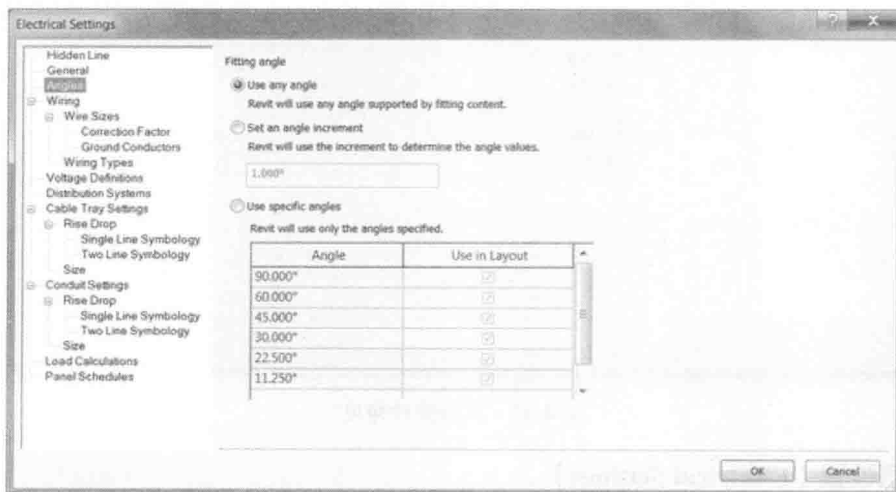


图 4-37 装配角

2. 其他电气设置（Other Electrical Settings）

ElectricalSetting 类支持访问的其他属性如下：

- Distribution System Types（分布系统类型）。
- Voltage Types（电压类型）。
- Wire Conduit Types（电缆桥架类型）。
- Wire Material Types（线材类型）。
- Wire Types（缆线类型）。

方法也可用于添加或从项目中删除分布系统类型、电压类型、线材类型和缆线类型。

4.3.6 布管系统配置（Routing Preferences）

布管系统配置可通过 RoutingPreferenceManager 类访问。该类的实例可从 MEPCurveType 类的属性获得。目前，只有 PipeType 和 DuctType 支持布管系统配置。

RoutingPreferenceManager 根据用户选择的标准，对选择管段类型、尺寸和管件类型的所有规则进行管理。RoutingPreferenceRule 类管理某管段或管件的系统配置，此类的实例可以添加到 RoutingPreferenceManager。每个布管系统配置规则依据所管理的布管项目类型进行分组。此类型由 RoutingPreferenceRuleGroupType 表示，包括表 4- 10 中的选项。

表 4-10 RoutingPreferenceRuleGroupType 类选项

成员名称	说 明
Undefined	未定义的组类型（默认初始值）
Segments	管段类型（e.g. pipe stocks）
Elbows	弯头类型
Junctions	接头类型（如 e.g.takeoff、tree、wye、tap）
Crosses	交叉类型
Transitions	过渡类型（注：多形状过渡可能有自己的分组）
Unions	联合类型，将两个管段连接在一起
MechanicalJoints	连接管件-管件、管段-管件、管段-管段的机械接点类型
TransitionsRectangularToRound	由矩形轮廓多形状过渡到圆形轮廓
TransitionsRectangularToOval	由矩形轮廓多形状过渡到椭圆形轮廓
TransitionsOvalToRound	由椭圆形轮廓多形状过渡到圆形轮廓

每个布管系统配置规则可以有一个或多个选择标准，由 RoutingCriterionBase 类及其派生类型 PrimarySizeCriterion 表示。PrimarySizeCriterion 根据最小、最大尺寸限制选择管件和管段。

RoutingConditions 类包含 RoutingCondition 实例集合。RoutingCondition 类表示布管信息，用作判定是否满足布管规则（如最小或最大直径）的输入。RoutingPreferencesManager.GetMEPPartId()方法根据 RoutingPreferenceRuleGroupType 和 RoutingConditions 获取管件或管段的 ID。

代码 4-33 示例了如何获取文件中所有的管道类型，并获取每个类型的布管系统配置管



理器，然后根据布管系统配置管理器中的规则获取每个管段的尺寸。

**代码 4-33: 使用布管系统配置**

```
private List<double> GetAvailablePipeSegmentSizesFromDocument (Document document)
{
    System.Collections.Generic.HashSet<double> sizes = new HashSet<double>();

    FilteredElementCollector collectorPipeType = new FilteredElementCollector (document);
    collectorPipeType.OfClass (typeof (PipeType));

    IEnumerable<PipeType> pipeTypes = collectorPipeType.ToElements().Cast<PipeType>();
    foreach (PipeType pipeType in pipeTypes)
    {
        RoutingPreferenceManager rpm = pipeType.RoutingPreferenceManager;

        int segmentCount = rpm.GetNumberOfRules (RoutingPreferenceRuleGroupType.Segments);
        for (int index = 0; index != segmentCount; ++index)
        {
            RoutingPreferenceRule segmentRule = rpm.GetRule (RoutingPreferenceRuleGroupType.Segments, index);
            Segment segment = document.GetElement (segmentRule.MEPPartId) as Segment;
            foreach (MEPSize size in segment.GetSizes())
            {
                sizes.Add (size.NominalDiameter); //Use a hash-set to remove duplicate sizes among Segments and
PipeTypes.
            }
        }
    }

    List<double> sizesSorted = sizes.ToList();
    sizesSorted.Sort();
    return sizesSorted;
}
```

# 第 5 章 进阶专题 (Advanced Topics)

## 5.1 在 Revit 模型中存储数据 (Storing Data in the Revit Model)

Revit API 提供了两种在 Revit 模型中存储数据的方法。第一种是使用共享参数。Revit API 提供了编程访问同一共享参数的功能，这些参数在 Revit 用户界面也可获得。共享参数若定义为可见，则它在图元的属性窗中是用户可见的。共享参数可以被指定到许多（但不是全部）图元类别。

另一种选择是可扩展存储，它允许您创建自定义的数据结构，然后指定其数据实例到模型中的图元。此数据在 Revit 用户界面中对用户总是不可见的，但当此类数据被定义后，通过其他第三方应用程序通过 Revit API 根据读/写访问分配模式访问。与 Shared Parameters 不同，可扩展存储不限于某些图元类别。可扩展存储的数据可以被指定到 Revit 模型基类 Element 派生的任何对象。

### 5.1.1 共享参数 (Shared Parameters)

Shared Parameters 是存储于外部文本文件的参数定义。定义由创建时生成的唯一标识符来标识，它可用于多个项目中。

本章介绍了如何通过 Revit 平台 API 访问共享参数。以下综述了如何获取共享参数并将其绑定到某些类别图元：

- 设置 SharedParametersFileName (共享参数文件名)。
- 获取 External Definition (外部定义)。
- Binding (绑定)。

### 5.1.2 定义文件 (Definition File)

DefinitionFile 对象表示一个共享参数文件。定义文件是一个普通文本文件。不要直接编辑定义文件，而是用 UI 或者 API 来编辑它。

#### 1. 定义文件格式 (Definition File Format)

共享参数定义文件是一个文本文件 (.txt)，有两个数据块：GROUP 和 PARAM。代码 5-1 是一个共享参数文本文件的示例。图 5-1 为编辑共享参数编辑界面。

代码 5-1：参数定义文件示例

```
# This is a Revit shared parameter file.
# Do not edit manually.
*GROUP ID NAME
GROUP 1 MyGroup
```





GROUP	2	AnotherGroup			
*PARAM	GUID	NAME	DATATYPE	DATACATEGORY	GROUP
PARAM	4b217a6d-87d0-4e64-9bbc-42b69d37dda6	MyParam	TEXT	1	1
PARAM	34b5cb95-a526-4406-806d-dae3e8c66fa9	Price	INTEGER	2	1
PARAM	05569bb2-9488-4be4-ae21-b065f93f7dd6	areaTags	FAMILYTYPE	-2005020	1

- GROUP 块包含将每个参数定义与组关联的成组条目。GROUP 块中有以下字段：
  - ID: 组及组中相关参数定义的唯一标识。
  - Name: 显示在 UI 中的组名称。
- PARAM 数据块包含参数定义。有以下字段：
  - GUID: 标识参数定义。
  - NAME: 参数定义名称。
  - DATATYPE: 参数类型。此字段可以是普通类型 (TEXT、INTEGER 等)、结构类型 (FORCE、MOMENT 等) 或普通的族类型 (Area Tags 等)。普通类型和结构类型参数可在文本文件中直接指定 (如 TEXT、FORCE)。若 DATATYPE 字段的值是 FAMILYTYPE, 则外加一个数字。如代码 5-2 中所示的 FAMILYTYPE 后面有 -2005020, 其表示族类型: Area Tags (图 5-2)。



图 5-1 编辑共享参数 (组及参数)



图 5-2 新参数定义

### 代码 5-2: 共享参数 FAMILYTYPE 例子

FAMILYTYPE -2005020

- GROUP: 一组 ID, 用于识别包含当前参数定义的组。
- VISIBLE: 标识参数是否可见。该字段值为 0 或 1, 0 = 不可见、1=可见。如定义文件示例所示, 有两个组:
  - MyGroup: ID 为 1, 包含 MyParam 参数定义, 为 Text 类型参数。
  - AnotherGroup: ID 为 2, 包含价格参数定义, 为 Integer 类型参数。



## 2. 定义文件访问 (Definition File Access)

附加代码 (add-in code) 中通过以下方法, 实现对定义文件的访问:

- 指定现有文本文件或新文件的 Autodesk.Revit.Options.Application.SharedParameters Filename 属性。
- 用 Application.OpenSharedParameterFile() 方法打开共享参数文件。
- 用 DefinitionFile.Groups 属性打开现有组或新组。
- 用 DefinitionGroup.Definitions 属性打开外部参数定义或创建新定义。

以下给出上述类和方法的更多信息:

- Autodesk.Revit.Parameters.DefinitionFile 类 —— DefinitionFile 对象, 表示共享参数文件。
  - 此对象包含若干 Group 对象。
  - 共享参数分成组以便于管理, 其中包含共享参数定义。
  - 根据需要可以添加新的定义。
  - DefinitionFile 对象可用 Application.OpenSharedParameterFile 方法检索。
- Autodesk.Revit.Parameters.ExternalDefinition 类 —— ExternalDefinition 类是 Definition 类的派生类。
  - ExternalDefinition 对象通过 DefinitionGroup 对象从共享参数文件创建。
  - 外部参数定义必须属于 Group, 它是共享参数定义的集合。
- Autodesk.Revit.Options.Application.SharedParametersFilename 属性 —— 用 Autodesk.Revit.Options.SharedParametersFilename 属性获取、设置共享参数文件路径。
  - 默认情况下, Revit 无共享参数文件。
  - 在使用之前, 初始化该属性。若未初始化则引发异常。
- Autodesk.Revit.Application.OpenSharedParameterFile 方法 —— 此方法返回一个表示 Revit 共享参数文件的对象。
  - Revit 一次使用一个共享参数文件。
  - 在 Revit Application Options 对象中设置共享参数文件的文件名称。若文件不存在则引发异常。

(1) 创建共享参数文件 (Create a Shared Parameter File)。由于共享参数文件是文本文件, 因此可以使用代码 5-3 创建或手动创建。

### 代码 5-3: 创建共享参数文件

```
private void CreateExternalSharedParamFile (string sharedParameterFile)
{
    System.IO.FileStream fileStream = System.IO.File.Create (sharedParameterFile);
    fileStream.Close();
}
```

(2) 访问现有共享参数文件 (Access an Existing Shared Parameter File)。由于可能有很多 Revit 共享参数文件, 因此要明确标识需要访问的文件和外部参数。以下两个步骤说明如何访问现有共享参数文件。



1) 从外部参数文件中获取定义文件 (Get DefinitionFile from an External Parameter File)。设置共享参数文件路径为 `app.Options.SharedParametersFilename`, 然后调用 `Autodesk.Revit.Application.OpenSharedParameterFile` 方法, 见代码 5-4。

#### 代码 5-4: 从外部参数文件中获取定义文件

```
private DefinitionFile SetAndOpenExternalSharedParamFile (  
    Autodesk.Revit.ApplicationServices.Application application, string sharedParameterFile)  
{  
    // set the path of shared parameter file to current Revit  
    application.Options.SharedParametersFilename = sharedParameterFile;  
    // open the file  
    return application.OpenSharedParameterFile();  
}
```

注: 设置共享参数路径时, 请考虑以下几点。

- 在每次安装期间, Revit 无法检测共享参数文件是否已由其他版本设置, 必须将共享参数文件重新绑定到新安装的 Revit。
- 如果 `Options.SharedParametersFilename` 被设置为不正确的路径, 只有在调用 `OpenSharedParameterFile` 时才会发生异常。
- Revit 可以使用多个共享参数文件。即使载入参数时只使用了一个参数文件, 当前文件也可以随意更改。

2) 遍历所有参数条目 (Traverse All Parameter Entries)。代码 5-5 演示了如何遍历参数条目并将结果显示在消息框中。

#### 代码 5-5: 遍历参数条目

```
private void ShowDefinitionFileInfo (DefinitionFile myDefinitionFile)  
{  
    StringBuilder fileInformation = new StringBuilder (500);  
    // get the file name  
    fileInformation.AppendLine ("File Name: " + myDefinitionFile.Filename);  
    // iterate the Definition groups of this file  
    foreach (DefinitionGroup myGroup in myDefinitionFile.Groups)  
    {  
        // get the group name  
        fileInformation.AppendLine ("Group Name: " + myGroup.Name);  
        // iterate the definitions  
        foreach (Definition definition in myGroup.Definitions)  
        {  
            // get definition name  
            fileInformation.AppendLine ("Definition Name: " + definition.Name);  
        }  
    }  
    TaskDialog.Show ("Revit", fileInformation.ToString());  
}
```

(3) 更改参数定义所属组 (Change the Parameter Definition Owner Group)。代码 5-6 演示了如何更改参数定义所属组。

**代码 5-6: 更改参数定义所属组**

```
private void ReadEditExternalParam (DefinitionFile file)
{
    // get ExternalDefinition from shared parameter file
    DefinitionGroups myGroups = file.Groups;
    DefinitionGroup myGroup = myGroups.get_Item ("MyGroup");
    if (null == myGroup)
        return;

    Definitions myDefinitions = myGroup.Definitions;
    ExternalDefinition myExtDef = myDefinitions.get_Item ("MyParam") as ExternalDefinition;
    if (null == myExtDef) return;

    StringBuilder strBuilder = new StringBuilder();
    // iterate every property of the ExternalDefinition
    strBuilder.AppendLine ("GUID: " + myExtDef.GUID.ToString());
    .AppendLine ("Name: " + myExtDef.Name)
    .AppendLine ("OwnerGroup: " + myExtDef.OwnerGroup.Name)
    .AppendLine ("Parameter Group" + myExtDef.ParameterGroup.ToString())
    .AppendLine ("Parameter Type" + myExtDef.ParameterType.ToString())
    .AppendLine ("Is Visible: " + myExtDef.Visible.ToString());

    TaskDialog.Show ("Revit", strBuilder.ToString());

    // change the OwnerGroup of the ExternalDefinition
    myExtDef.OwnerGroup = myGroups.get_Item ("AnotherGroup");
}
```

**5.1.3 绑定 (Binding)**

请在插件代码中完成以下步骤，以绑定指定参数：

- (1) 使用 InstanceBinding 或 TypeBinding 对象，新建包括参数绑定类别的 Binding 对象。
- (2) 使用 Document.ParameterBindings 对象将绑定和定义添加到文件。

以下给出上述类和方法的更多信息：

- Autodesk.Revit.Parameters.BindingMap 类：从 Document.ParameterBindings 属性中检索到的 BindingMap 对象。
  - 参数绑定将参数定义与一个或多个类别中的图元关联。
  - 映射用于查询现有绑定，此外，Insert 方法用于生成新的参数绑定。
- Parameters.BindingMap.Insert (Definition, Binding) 方法：绑定对象类型决定该参数是与所有实例绑定还是只与类型绑定。
  - 参数定义不能与实例和类型同时绑定。
  - 如果绑定参数已存在，则该方法返回 false。

**1. 类型绑定 (Type Binding)**

Autodesk.Revit.Parameters.TypeBinding 对象用于将属性绑定于 Revit 类型，如图 5-3 参数属性对话框中墙类型的绑定。它不同于实例绑定，因为属性是由类型绑定所识别出的



全部实例所共享。更改某个类型的参数会影响到所有相同类型的实例。

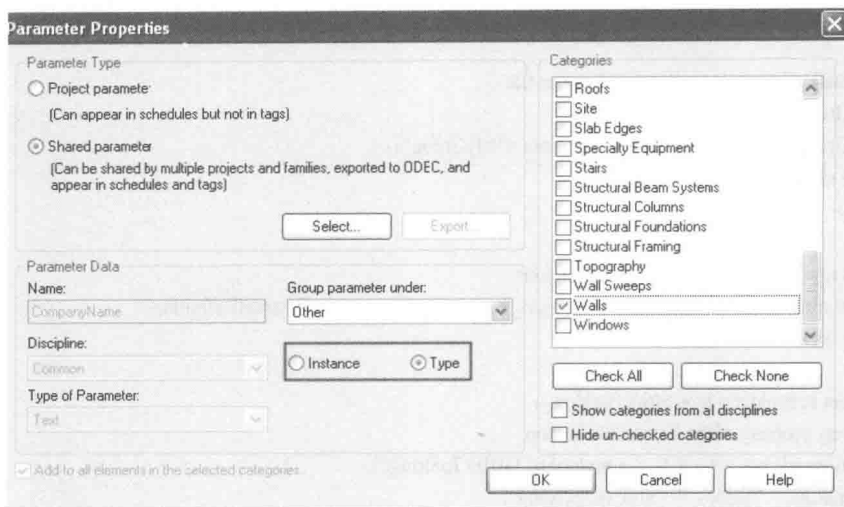


图 5-3 参数属性对话框中的类型绑定

代码 5-7 演示了如何使用共享参数文件添加参数定义。该代码和使用图 5-3 对话框执行的是同样操作。按以下顺序创建参数定义：

- (1) 创建一个共享参数文件。
- (2) 创建一个墙类型的定义组以及参数定义。
- (3) 根据墙类别，在当前文件中将定义绑定到墙类型。

#### 代码 5-7：使用共享参数文件添加类型参数定义

```
public bool SetNewParameterToTypeWall (UIApplication app, DefinitionFile myDefinitionFile)
{
    // Create a new group in the shared parameters file
    DefinitionGroups myGroups = myDefinitionFile.Groups;
    DefinitionGroup myGroup = myGroups.Create ("MyParameters");

    // Create a type definition
    Definition myDefinition_CompanyName =
        myGroup.Definitions.Create ("CompanyName", ParameterType.Text);

    // Create a category set and insert category of wall to it
    CategorySet myCategories = app.Application.Create.NewCategorySet ();
    // se BuiltInCategory to get category of wall
    Category myCategory = app.ActiveUIDocument.Document.Settings.Categories.get_Item (BuiltInCategory.OST_Walls);
    myCategories.Insert (myCategory);

    //Create an object of TypeBinding according to the Categories
    TypeBinding typeBinding = app.Application.Create.NewTypeBinding (myCategories);

    // Get the BindingMap of current document
    BindingMap bindingMap = app.ActiveUIDocument.Document.ParameterBindings;
```



```
// Bind the definitions to the document
bool typeBindOK = bindingMap.Insert (myDefinition_CompanyName, typeBinding,
    BuiltInParameterGroup.PG_TEXT);
return typeBindOK;
}
```

## 2. 实例绑定 ( Instance Binding )

Autodesk. Revit. Parameters. InstanceBinding 对象表示在某个参数定义与某类别的实例参数之间的绑定。图 5-4 所示对话框说明了 Walls 类型的绑定为实例绑定。

一旦绑定，该参数会显示在该实例的所有属性对话框中。更改任意某个实例的参数，不会导致任何其他实例中参数值的变化。

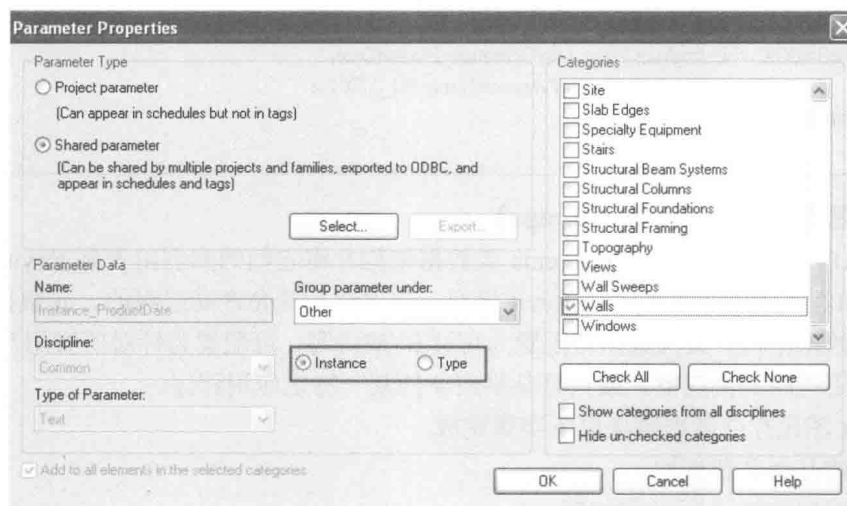


图 5-4 参数属性对话框中的实例绑定

代码 5-8 演示了如何使用共享参数文件添加参数定义。按以下顺序添加参数定义：

- (1) 创建一个共享参数文件。
- (2) 创建一个定义组以及所有的 Walls 实例的定义。
- (3) 根据墙类别，在当前文件中将定义绑定到每个墙实例参数。

### 代码 5-8：使用共享参数文件添加实例参数定义

```
public bool SetNewParameterToInstanceWall (UIApplication app, DefinitionFile myDefinitionFile)
{
    // create a new group in the shared parameters file
    DefinitionGroups myGroups = myDefinitionFile.Groups;
    DefinitionGroup myGroup = myGroups.Create ("MyParameters1");

    // create an instance definition in definition group MyParameters
    Definition myDefinition_ProductDate =
        myGroup.Definitions.Create ("Instance_ProductDate", ParameterType.Text);
}
```



```
// create a category set and insert category of wall to it
CategorySet myCategories = app.Application.Create.NewCategorySet( );
// use BuiltInCategory to get category of wall
Category myCategory = app.ActiveUIDocument.Document.Settings.Categories.get_Item (
    BuiltInCategory.OST_Walls);
myCategories.Insert (myCategory);

//Create an instance of InstanceBinding
InstanceBinding instanceBinding =
    app.Application.Create.NewInstanceBinding (myCategories);

// Get the BindingMap of current document.
BindingMap bindingMap = app.ActiveUIDocument.Document.ParameterBindings;

// Bind the definitions to the document
bool instanceBindOK = bindingMap.Insert (myDefinition_ProductDate,
    instanceBinding, BuiltInParameterGroup.PG_TEXT);
return instanceBindOK;
}
```

#### 5.1.4 可扩展存储 (Extensible Storage)

Revit API 允许创建自己的 Schema 类数据结构并将它们的实例附着到 Revit 模型中的任何 Element。基于架构的数据随 Revit 模型一起保存，并允许高层级的、元数据增强的、面向对象的数据结构。架构数据可配置为可读和/或可写，可配置为针对所有用户，或只是某一特定应用程序的供应商，或只是某供应商的某一特定应用程序。

随 Revit 图元存储数据需按以下步骤完成：

- (1) 创建并命名新架构。
- (2) 设置架构的读/写访问权限。
- (3) 定义架构的一个或多个数据字段。
- (4) 根据架构创建条目。
- (5) 对条目的字段赋值。
- (6) 将条目关联到 Revit 图元。

##### 1. 架构和架构构造器 (Schemas and SchemaBuilder)

创建可扩展存储的第一步是定义架构。架构类似于面向对象的编程语言中的类。使用 SchemaBuilder 类的构造函数来创建新的架构。SchemaBuilder 是个用于创建架构的 helper 类。一旦 SchemaBuilder 完成了架构，即可使用 Schema 类访问架构属性。到了这个阶段，架构就不再可编辑。

虽然 SchemaBuilder 构造函数采用 GUID (全局唯一标识符) 来识别架构，但还是需要一个架构名称。在创建架构之后，调用 SchemaBuilder.SetSchemaName() 指定一个比较好用的架构名称标识符。架构名称可用于在错误信息中找到该架构。

架构相关实体段的读取和写入访问权限级别可以独立设置。选项为 Public (公用)、Vendor (供应商) 或 Application (应用程序)。如果读取或写入访问权限级别设置为 Vendor，那么必须指定可访问架构实体的第三方供应商的 VendorId (供应商识别号)。若访问级别设



置为 **Application**，则必须提供可访问架构实体的应用程序或插件的 **GUID**。

注：架构是随文件一起存储的，且任何 **Revit API** 插件都可读取文件中的可用架构，以及某些架构数据。然而，基于架构中定义的读访问权限，对架构字段的访问是受此限制的，并且，若定义了架构的读、写访问权限设置，则存储于特定图元的实体中的实际数据也受此限制。

## 2. 字段和字段构造器 (Fields and FieldBuilder)

一旦创建架构之后，就可定义字段。字段类似于类的属性。它包含名称、文档、值类型和单位类型。字段可以是简单类型、数组或映射。系统允许的简单数据类型见表 5-1。

表 5-1 数据类型

类型	默认值
int	0
short	0
byte	0
double	0.0
float	0.0
bool	false
string	空字符串 ("")
GUID	Guid.Empty {00000000-0000-0000-0000-000000000000}
ElementId	ElementId.InvalidElementId
Autodesk.Revit.DB.XYZ	(0.0, 0.0, 0.0)
Autodesk.Revit.DB.UV	(0.0, 0.0)

此外，字段可以是 **Autodesk.Revit.DB.ExtensibleStorage.Entity** 类型。换言之，是另一种 **Schema** 的实例，也就是 **SubSchema** 或 **SubEntity**。这种类型的字段的默认值为空架构的 **Entity** 和 **Guid.Empty** 的标识符。

可以使用 **SchemaBuilder.AddSimpleField method()** 方法创建一个简单的字段来指定字段的名称和类型。**AddSimpleField()** 返回一个 **FieldBuilder**，这是一个用于定义 **Fields** 的 helper 类。如果指定字段的类型为 **Entity**，则使用 **FieldBuilder.SetSubSchemaGUID()** 来指定要存储在该字段实体的架构 **GUID**。

使用 **SchemaBuilder.AddArrayField()** 方法在 **Schema** 中创建一个含有一组值的字段，其带有给定名称和所含值的类型。数组字段的类型可以与简单字段完全相同。

使用 **SchemaBuilder.AddMapField()** 方法在架构中创建一个包含有序的 **key-value** (键-值) 映射的字段，带有给定名称、“键”类型和所含值的类型。受支持的值类型与简单字段相同。支持的“键”类型仅限于整型、短整型、字节、字符串、布尔型、**ElementId** 和 **GUID**。

一旦用 **SchemaBuilder** 完成了架构，字段就不再可用 **FieldBuilder** 进行编辑。在这个阶段，用 **Schema** 类方法，通过名称或架构中定义的所有字段的列表来获取字段。





### 3. 实体 (Entities)

在架构所有字段都已定义之后, `SchemaBuilder.Finish()` 会返回已完成的 `Schema`。可以使用该架构创建新 `Entity`。架构中的每个字段, 其值都可以使用 `Entity.Set()` 存储, 它采用一个 `Field` 和一个值, 其类型取决于字段类型。一旦实体的所有适用字段都已设置完成, 即可使用 `Element.SetEntity()` 方法将它指定给某个图元。

随后, 要检索数据, 调用 `Element.GetEntity()` 传入相应 `Schema`。如果没有基于架构随 `Element` 一同保存的实体, 则将返回一个无效的 `Entity`。要核查返回的 `Entity` 是否有效, 请调用 `Entity.IsValid()` 方法。使用 `Entity.Get()` 方法可以获取实体的 `Field` 值。

要确定是否有储入图元的实体, 请使用 `Element.GetEntitySchemaGuids()` 方法, 返回实体中任意图元的 `Schema` 标识符。它可用于静态方法 `Schema.Lookup()` 来检索相应的 `Schemas`。

代码 5-9 把所有这些步骤集成到了一起。

#### 代码 5-9: 可扩展存储

```
// Create a data structure, attach it to a wall, populate it with data, and retrieve the data back from the wall
void StoreDataInWall (Wall wall, XYZ dataToStore)
{
    Transaction createSchemaAndStoreData = new Transaction (wall.Document,
        "tCreateAndStore");
    createSchemaAndStoreData.Start();
    SchemaBuilder schemaBuilder =
        new SchemaBuilder (new Guid ("720080CB-DA99-40DC-9415-E53F280AA1F0"));
    schemaBuilder.SetReadAccessLevel (AccessLevel.Public); // allow anyone to read the object
    schemaBuilder.SetWriteAccessLevel (AccessLevel.Vendor); // restrict writing to this vendor only
    schemaBuilder.SetVendorId ("ADSK"); // required because of restricted write-access
    schemaBuilder.SetSchemaName ("WireSpliceLocation");
    // create a field to store an XYZ
    FieldBuilder fieldBuilder =
        schemaBuilder.AddSimpleField ("WireSpliceLocation", typeof (XYZ));
    fieldBuilder.SetUnitType (UnitType.UT_Length);
    fieldBuilder.SetDocumentation ("A stored location value representing a wiring splice in a wall.");

    Schema schema = schemaBuilder.Finish(); //register the Schema object
    Entity entity = new Entity (schema); //create an entity (object) for this schema (class)
    // get the field from the schema
    Field fieldSpliceLocation = schema.GetField ("WireSpliceLocation");
    // set the value for this entity
    entity.Set<XYZ> (fieldSpliceLocation, dataToStore, DisplayUnitType.DUT_METERS);
    wall.SetEntity (entity); // store the entity in the element

    // get the data back from the wall
    Entity retrievedEntity = wall.GetEntity (schema);
    XYZ retrievedData =
        retrievedEntity.Get<XYZ> (schema.GetField ("WireSpliceLocation"),
            DisplayUnitType.DUT_METERS);
    createSchemaAndStoreData.Commit();
}
```



#### 4. 可扩展存储的优点 (Extensible Storage Advantages)

- 自说明和自定义 (Self Documenting and Self-Defining)。
- 通过添加字段、单位、子实体, 描述字符串来创建架构, 这不仅是用于存储数据的一种手段, 它还是为其他用户提供的隐式说明文件, 以及为其他人以后创建相同架构的实体提供的捷径方式。
- 获取“局部性”优点 (Takes Advantage of Locality)。

由于架构实体以每个图元的方式来存储, 因此, 当应用程序可能只需要当前所选梁的数据时, 不用读取文件中所有扩展存储数据 (如所有梁族实例的所有数据)。一般来说, 这可能使得数据访问代码更明确、更有针对性, 以获得更好的数据访问性能。

## 5.2 事务 (Transactions)

事务是个类似上下文的对象, 它封装了 Revit 模型中的任何更改。对文件的任何更改只能在该文件的活动事务打开之际进行。试图在事务之外更改文件将引发异常。所作更改不会成为模型的一部分, 直到活动事务提交完成。因此, 在事务中所作的所有更改都可 (通过析构函数) 显式或隐式回滚。在任何时候, 每个文件只能打开一个事务。一个事务可以包括多个操作。

Revit API 中与事务相关的主类有三个:

- Transaction (事务)。
- SubTransaction (子事务)。
- TransactionGroup (事务组)。

本节将更深入地讨论这些类。对文件进行更改只需要 Transaction 类。而其他的类用于更好地组织更改。要注意的是, 在调用该命令时, 应用于 IExternalCommand 定义的 TransactionMode 属性, 会影响 Revit 预期处理事务的方式。回顾事务属性可获得更多信息。

注: 若从线程以外或非模态对话框以外启动事务, 则将引发异常。事务只能从受支持的 API 工作流中启动, 如部分外部命令、事件、更新器或者回调。

### 5.2.1 事务类 (Transaction Classes)

所有三个事务对象所共享的通用方法见表 5-2。

表 5-2 通用事务对象方法

方 法	说 明
Start	启动上下文
Commit	终止上下文并提交对文件所作更改
Rollback	终止上下文并放弃对文件所作更改
GetStatus	返回事务对象的当前状态

除了 GetStatus() 方法可返回当前状态, Start、Commit 和 RollBack 方法也会返回一个



TransactionStatus 用于说明该方法是否成功。可用的 TransactionStatus 值见表 5-3。

表 5-3 事务状态值

状态	说 明
Uninitialized	对象实例化后的初始值，但上下文尚未启动
Started	事务对象已成功启动（Start 方法已调用）
RolledBack	事务对象成功回滚（Rollback 方法已调用）
Committed	事务对象成功提交完成（Commit 方法已调用）
Pending	试图提交或回滚事务对象，但因故障而进程尚无法完成，正等待最终用户的响应（在非模态对话框中）。一旦故障处理完成，状态将自动更新（到 Committed 或 RolledBack 状态）

### 1. 事务 (Transaction)

事务是便于 Revit 模型作任意更改所需的上下文。一次只能打开一个事务；不允许嵌套使用。每个事务都须有个名称，代码 5-10 演示了创建草图过程中如何创建事务。

#### 代码 5-10：使用事务

```
public void CreatingSketch (UIApplication uiApplication)
{
    Autodesk.Revit.DB.Document document = uiApplication.ActiveUIDocument.Document;
    Autodesk.Revit.ApplicationServices.Application application = uiApplication.Application;

    // Create a few geometry lines. These lines are transaction (not in the model),
    // therefore they do not need to be created inside a document transaction.
    XYZ Point1 = XYZ.Zero;
    XYZ Point2 = new XYZ (10, 0, 0);
    XYZ Point3 = new XYZ (10, 10, 0);
    XYZ Point4 = new XYZ (0, 10, 0);

    Line geomLine1 = Line.CreateBound (Point1, Point2);
    Line geomLine2 = Line.CreateBound (Point4, Point3);
    Line geomLine3 = Line.CreateBound (Point1, Point4);

    // This geometry plane is also transaction and does not need a transaction
    XYZ origin = XYZ.Zero;
    XYZ normal = new XYZ (0, 0, 1);
    Plane geomPlane = application.Create.NewPlane (normal, origin);

    // In order to a sketch plane with model curves in it, we need
    // to start a transaction because such operations modify the model.
    // All and any transaction should be enclosed in a "using"
    // block or guarded within a try-catch-finally blocks
    // to guarantee that a transaction does not out-live its scope.
    using (Transaction transaction = new Transaction (document))
    {
        if (transaction.Start ("Create model curves") == TransactionStatus.Started)
        {

```



```
// Create a sketch plane in current document
SketchPlane sketch = SketchPlane.Create (document, geomPlane);

// Create a ModelLine elements using the geometry lines and sketch plane
ModelLine line1 = document.Create.NewModelCurve (geomLine1, sketch) as ModelLine;
ModelLine line2 = document.Create.NewModelCurve (geomLine2, sketch) as ModelLine;
ModelLine line3 = document.Create.NewModelCurve (geomLine3, sketch) as ModelLine;

// Ask the end user whether the changes are to be committed or not
TaskDialog taskDialog = new TaskDialog ("Revit");
taskDialog.MainContent = "Click either [OK] to Commit, or [Cancel] to Roll back the transaction.";
TaskDialogCommonButtons buttons = TaskDialogCommonButtons.Ok | TaskDialogCommonButtons.Cancel;
taskDialog.CommonButtons = buttons;

if (TaskDialogResult.Ok == taskDialog.Show())
{
    // For many various reasons, a transaction may not be committed
    // if the changes made during the transaction do not result a valid model.
    // If committing a transaction fails or is canceled by the end user,
    // the resulting status would be RolledBack instead of Committed.
    if (TransactionStatus.Committed != transaction.Commit())
    {
        TaskDialog.Show ("Failure", "Transaction could not be committed");
    }
}
else
{
    transaction.Rollback();
}
}
```

## 2. 子事务 (SubTransaction)

**SubTransaction** 可用于封入一组“模型-修改”操作。**Sub-transactions** 是个可选项。修改模型并不一定需要它们。它们是个便利工具，可逻辑上将任务化大为小。子事务只能在一个已经开启的事务内创建，且必须在该事务结束（提交或回滚）之前结束（提交或回滚）。与事务不同，子事务可以嵌套，但任何嵌套的子事务必须在封装的子事务结束前结束。子事务没有名称，因为它们不显示在 Revit 撤销菜单上。

## 3. 事务组 (TransactionGroup)

**TransactionGroup** 允许将几个独立的事务一起成组，这使得组的所有者有机会一次处理多个事务。当事务组将要结束时，可以回滚，意味着所有先前提交的属于该组的事务都将回滚。如果不回滚，则事务组可被提交或“融合”。前一种情况下，（组内）提交的所有事务将脱离事务组。后一种情况下，组内的事务将合并成带有该组名称的单个事务。

事务组只能在没有其他开启的事务时启动，且必须仅在所有封装的事务结束（回滚或提交）之后结束。事务组可以嵌套，但任何嵌套组必须在封装的事务组结束之前结束。事



务组是个可选项。对模型进行修改并不一定需要它们。

代码 5-11 演示了如何在 TransactionGroup 中使用 Assimilate() 方法来合并两个单独的事务。代码 5-11 结果是将带有 “Level and Grid” 名称的单个撤销项添加到撤销菜单。

#### 代码 5-11: 在事务组中合并多个事务

```
public void CompoundOperation (Autodesk.Revit.DB.Document document)
{
    // All and any transaction group should be enclosed in a 'using' block or guarded within
    // a try-catch-finally blocks to guarantee that the group does not out-live its scope.
    using (TransactionGroup transGroup = new TransactionGroup (document, "Level and Grid"))
    {
        if (transGroup.Start() == TransactionStatus.Started)
        {
            // We are going to call two methods, each having its own local transaction.
            // For our compound operation to be considered successful, both the individual
            // transactions must succeed. If either one fails, we will roll our group back,
            // regardless of what transactions might have already been committed.

            if (CreateLevel (document, 25.0) && CreateGrid (document, new XYZ (0, 0, 0), new XYZ (10, 0, 0)))
            {
                // The process of assimilating will merge the two (or any number of) committed
                // transaction together and will assign the grid's name to the one resulting transaction,
                // which will become the only item from this compound operation appearing in the undo menu.
                transGroup.Assimilate();
            }
            else
            {
                // Since we could not successfully finish at least one of the individual
                // operation, we are going to roll the entire group back, which will
                // undo any transaction already committed while this group was open.
                transGroup.Rollback();
            }
        }
    }
}

public bool CreateLevel (Autodesk.Revit.DB.Document document, double elevation)
{
    // All and any transaction should be enclosed in a 'using'
    // block or guarded within a try-catch-finally blocks
    // to guarantee that a transaction does not out-live its scope.
    using (Transaction transaction = new Transaction (document, "Creating Level"))
    {
        // Must start a transaction to be able to modify a document

        if (TransactionStatus.Started == transaction.Start())
        {
            if (null != document.Create.NewLevel (elevation))
            {
            }
        }
    }
}
```



```

        // For many various reasons, a transaction may not be committed
        // if the changes made during the transaction do not result a valid model.
        // If committing a transaction fails or is canceled by the end user,
        // the resulting status would be RolledBack instead of Committed.
        return ( TransactionStatus.Committed == transaction.Commit() );
    }

    // For we were unable to create the level, we will roll the transaction back
    // (although on this simplified case we know there weren't any other changes)

    transaction.Rollback();
}
}
return false;
}

public bool CreateGrid (Autodesk.Revit.DB.Document document, XYZ p1, XYZ p2)
{
    // All and any transaction should be enclosed in a 'using'
    // block or guarded within a try-catch-finally blocks
    // to guarantee that a transaction does not out-live its scope.
    using (Transaction transaction = new Transaction (document, "Creating Grid"))
    {
        // Must start a transaction to be able to modify a document
        if (TransactionStatus.Started == transaction.Start())
        {
            // We create a line and use it as an argument to create a grid
            Line gridLine = Line.CreateBound (p1, p2);

            if ((null != gridLine) && (null != document.Create.NewGrid (gridLine)))
            {
                if (TransactionStatus.Committed == transaction.Commit())
                {
                    return true;
                }
            }

            // For we were unable to create the grid, we will roll the transaction back
            // (although on this simplified case we know there weren't any other changes)

            transaction.Rollback();
        }
    }
    return false;
}
}

```

### 5.2.2 事件中的事务 ( Transactions in Events )

#### 1. 在事件过程中修改文件 ( Modifying the Document during an Event )

事件不会自动开启事务。因此，在事件过程中，文件将不会被修改，除非事件处理程



序中的某个程序,通过在一个事务内作更改来修改它。如果事件处理程序根据需要开启了一个事务,则也必须关闭它(提交或回滚),否则所有更改都将丢失。

要注意的是,在某些事件(如 DocumentClosing 事件)过程中,活动文件是不允许修改的。在这样的事件过程中,如果事件处理程序试图修改,则将引发异常。该事件的说明文件会指出该事件是否是只读的。

## 2. 文件已修改事件 (DocumentChanged Event)

每个事务提交、撤销或重做之后,会引发 DocumentChanged 事件。这是个只读事件,旨在保持外部数据与 Revit 数据库处于同步状态。若要更新 Revit 数据库以响应图元的更改,则请使用动态模型更新框架。

### 5.2.3 故障处理选项 (Failure Handling Options)

故障处理选项是在事务结束时,若有任何故障的话应该如何处理的选项。故障处理选项可以在调用 Transaction.Commit()或 Transaction.Rollback()之前的任何时候,使用 Transaction.SetFailureHandlingOptions()方法进行设置。不过,在事务提交或回滚之后,所有选项都会返回到各自的默认设置。

SetFailureHandlingOptions()方法采用 FailureHandlingOptions 对象作为参数。无法创建此对象,必须使用 GetFailureHandlingOptions()方法从事务中获得。选项通过调用相应的 Set 方法如 SetClearAfterRollback()来设置。以下各节将更详细地讨论故障处理选项。

#### 1. 回滚后清除 (ClearAfterRollback)

此选项控制是否在事务回滚后清除所有警告。默认值为 false。

#### 2. 延迟小警告 (DelayedMiniWarnings)

如果有任何“小”警告的话,此选项控制它是否在当前结束的事务末显示,或是否延迟到下一个事务结束时显示。这通常用于一连串的事务,当不需要在每一步结束都显示中间警告,而是等到整个事务链完成再显示。

对于多个事务,警告可能会被延迟。未将此选项设置为 true 的第一个事务会显示自身所有警告(如果有的话),而且可能会从先前的事务累积所有警告。此选项默认值为 false。

注:此选项在模态模式中会被忽略(请参阅下节“强制模态处理”)。

#### 3. 强制模态处理 (ForcedModalHandling)

此选项控制对最终故障进行模态处理还是非模态处理。默认值为 true。请注意,如果设置为非模态故障处理,处理事务可能非同步进行,这意味着在 Commit 或 RollBack 调用返回后,该事务仍未完成(状态为“Pending”)。

#### 4. 设置故障预处理器 (SetFailuresPreprocessor)

若提供了此接口,则当发现故障后,在事务结束时被调用。预处理程序可检查甚至尝试解决当前故障。请参阅 5.8 节故障发布和处理以获得更多信息。

#### 5. 设置事务终结器 (SetTransactionFinalizer)

终结器是个接口,而如果提供,则可用于在事务结束时执行自定义操作。要注意的是,它不是在调用 Commit()或 RollBack()方法时被调用,而仅在提交或回滚过程完成之后。事务终结器必须实现 ITransactionFinalizer 接口,这就需要定义两个函数:



- OnCommitted: 在事务提交结束时调用。
- OnRolledBack: 在事务回滚结束时调用。

注: 由于调用终结器是在事务完成之后, 因此终结器不可修改该文件, 除非启动一个新事务。

### 5.2.4 获取图元几何和分析模型 (Getting Element Geometry and AnalyticalModel)

在创建新图元或修改图元之后, 传播整个模型的变化需要图元重生成和自动连接。若没有重生成 (以及自动连接, 当相关时), 图元的 Geometry 属性和 AnalyticalModel 要么是不可获取的 (在创建新图元的情况下), 要么可能是无效的。重要的是在访问图元的几何或分析模型之前, 应了解重生成如何及何时发生。

虽然传播模型中的更改需要重生成和自动连接, 但这可能会非常耗时。因此, 这些事件最好只在必需时发生。

当修改模型的事务成功提交, 或 Document.Regenerate() 抑或 Document.AutoJoinElements() 方法被调用, 都会自动发生重生成和自动连接。Regenerate() 和 AutoJoinElements() 只能在开启的事务内部调用。应当指出的是, Regenerate() 方法可能会失败, 在这种情况下, 将引发 RegenerationFailedException。如果发生这种情况, 则需通过回滚当前事务或子事务来对文件的更改作回滚处理。

关于 AnalyticalModel 对象的更多信息, 请参阅 4.2 节 Revit Structure 中分析模型。关于 Geometry 属性的更多信息, 请参阅 3.7 节几何。

代码 5-12 演示了如何用事务填充这些属性。

#### 代码 5-12: 用事务填充几何和分析模型属性

```
public void TransactionDuringElementCreation (UIApplication uiApplication, Level level)
{
    Autodesk.Revit.DB.Document document = uiApplication.ActiveUIDocument.Document;

    // Build a location line for the wall creation
    XYZ start = new XYZ (0, 0, 0);
    XYZ end = new XYZ (10, 10, 0);
    Autodesk.Revit.DB.Line geomLine = Line.CreateBound (start, end);

    // All and any transaction should be enclosed in a 'using'
    // block or guarded within a try-catch-finally blocks
    // to guarantee that a transaction does not out-live its scope.
    using (Transaction wallTransaction = new Transaction (document, "Creating wall"))
    {
        // To create a wall, a transaction must be first started
        if (wallTransaction.Start() == TransactionStatus.Started)
        {
            // Create a wall using the location line
            Wall wall = Wall.Create (document, geomLine, level.Id, true);

            // the transaction must be committed before you can
            // get the value of Geometry and AnalyticalModel.
        }
    }
}
```





```
if (wallTransaction.Commit() == TransactionStatus.Committed)
{
    Autodesk.Revit.DB.Options options = uiApplication.Application.Create.NewGeometryOptions();
    Autodesk.Revit.DB.GeometryElement geoelem = wall.get_Geometry (options);
    Autodesk.Revit.DB.Structure.AnalyticalModel analyticalmodel = wall.GetAnalyticalModel();
}
}
```

此示例的事务时间轴见图 5-5。

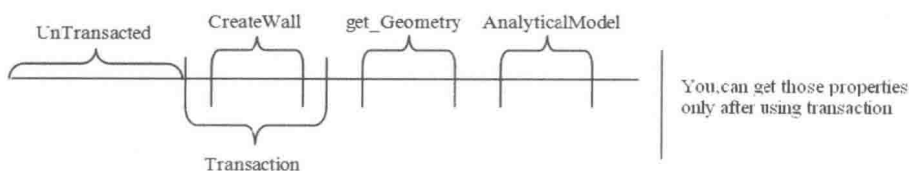


图 5-5 事务时间轴

### 5.2.5 临时事务 (Temporary Transactions)

事务并不总是需要提交，事务框架还允许回滚事务。在事务处理过程中出错时，这是很有用的，而且可以直接用此技术创建临时事务。

使用临时事务可用于某些类型的分析。例如，若应用程序要提取洞口切割之前的墙或其他对象几何属性，则应连同 `Document.Delete()` 一起使用临时事务。当应用程序删除切割目标图元的那个图元时，被切割图元的几何形状会恢复到其原始状态(在文件重生成之后)。

要使用临时事务：

- (1) 使用事务构造函数实例化事务，并为其命名。
- (2) 调用 `Transaction.Start()`。
- (3) 对文件进行临时更改（图元修改、删除或创建）。
- (4) 重生成文件。
- (5) 提取所需几何形状和属性。
- (6) 调用 `Transaction.RollBack()` 以恢复文件为原先状态。

此技术也适用于子事务。

## 5.3 事件 (Events)

事件是 Revit 用户界面或 API 工作流中的具体操作触发的通知。通过订阅事件，当某项操作即将发生或刚刚发生，并执行某些与该事件相关的操作时，可以通知插件应用程序。某些事件会成对出现于操作前后，其中一个事件在操作之前发生（“前置”事件），而另一个在操作之后发生（“后置”事件）。不以前置/后置成对发生的事件称为“单一”事件。

Revit 提供了对应用程序级（如 `ApplicationClosing` 或 `DocumentOpened`）和文件级（如 `DocumentClosing` 和 `DocumentPrinting`）事件的访问。同样应用程序级事件可从 `Application` 类



获得,也可从 `ControlledApplication` 类获得,它代表无访问文件的 Revit 应用。`ControlledApplication` `OnStartup()`和 `OnShutdown()`方法的加载项。就事件的订阅和取消订阅而言,这些类是可以互换的;从 `ControlledApplication` 类订阅事件等同于从 `Application` 类中订阅。

事件还可以被归类为数据库(DB)事件或用户界面(UI)事件。DB 事件可由 `Application` 和 `Document` 类获得,而 UI 事件可由 `UIApplication` 类获得(目前所有 UI 事件仅限于应用程序级别)。

某些事件被视为只读事件,这意味着在执行期间不得修改模型。事实上,API 帮助文件中所记录的就是只读事件。重要的是,要知道即使在常规事件(即非只读事件)过程中,模型也可能处于不能编辑的状态。程序员应检查 `Document.IsModifiable` 和 `Document.IsReadOnly` 属性,以确定该模型是否可被修改。

### 5.3.1 数据库事件 (Database Events)

表 5-4 列出了数据库事件及其类型,以及是否可在应用程序级和/或文件级使用。

表 5-4 数据库事件及其类型

事件	类型	应用程序	文档	事件说明
<code>DocumentChanged</code>	single	X		事务提交、撤销或重新执行时通知
<code>DocumentClosing</code>	pre	X	X	Revit 即将关闭文件时通知
<code>DocumentClosed</code>	post	X		Revit 文件关闭完成时通知
<code>DocumentCreating</code>	pre	X		Revit 即将创建新文件时通知
<code>DocumentCreated</code>	post	X		Revit 新文件创建完成时通知
<code>DocumentOpening</code>	pre	X		Revit 即将打开文件时通知
<code>DocumentOpened</code>	post	X		Revit 文件打开后通知
<code>DocumentPrinting</code>	pre	X	X	Revit 即将打印文件的视图或视图集时通知
<code>DocumentPrinted</code>	post	X	X	Revit 文件的视图或视图集打印完成时通知
<code>DocumentSaving</code>	pre	X	X	Revit 即将存储文件时通知
<code>DocumentSaved</code>	post	X	X	Revit 文件存储完成时通知
<code>DocumentSavingAs</code>	pre	X	X	Revit 即将以新名称存储文件时通知
<code>DocumentSavedAs</code>	post	X	X	Revit 文件以新名称存储完成时通知
<code>DocumentSynchronizingWithCentral</code>	pre	X		Revit 文件即将与中心文件同步时通知
<code>DocumentSynchronizedWithCentral</code>	post	X		Revit 文件与中心文件同步完成时通知
<code>FailuresProcessing</code>	single	X		Revit 事务结束前正处理故障时通知
<code>FileExporting</code>	pre	X		Revit 即将导出受 API 支持的格式文件时通知
<code>FileExported</code>	post	X		Revit 导出受 API 支持的格式文件完成时通知
<code>FileImporting</code>	pre	X		Revit 即将导入受 API 支持的格式文件时通知
<code>FileImported</code>	post	X		Revit 导入受 API 支持的格式文件完成时通知
<code>ProgressChanged</code>	single	X		Revit 操作有进度指示数据时通知
<code>ViewPrinting</code>	pre	X	X	Revit 即将打印文件的视图时通知
<code>ViewPrinted</code>	post	X	X	Revit 文件视图打印完成时通知



当 Revit 文件完成更改后触发 `DocumentChanged` 事件。每当 Revit 事务提交、撤销或重新执行时引发此事件。这是个只读事件，旨在让外部数据与 Revit 数据库保持同步状态。要更新 Revit 数据库以响应图元更改，请使用 `IUpdater` 框架。

`DocumentChanged` 事件会使用 `DocumentChangedEventArgs` 类。该类有几个方法，用来获取任何新添加图元 [`GetAddElementIds()`]、被删除图元 [`GetDeletedElementIds()`] 或已修改图元 [`GetModifiedElementIds()`] 的图元 ID。`GetAddElementIds()` 和 `GetModifiedElementIds()` 方法可用带有 `ElementFilter` 的重载，这使它易于仅检测所关心的更改。

### 5.3.2 用户界面事件 (User Interface Events)

表 5-5 列出了用户界面事件及其类型，以及是否可在应用程序级和/或文件级使用。

表 5-5 用户界面事件及其类型

事件	类型	UI 应用程序	ControlledApplication	事件说明
ApplicationClosing	pre	X		Revit 应用程序即将关闭时通知
ApplicationInitialized	single		X	在 Revit 应用程序完成初始化、所有外部应用程序已启动且应用程序已准备好处理文件时通知
DialogBoxShowing	single	X		当 Revit 正显示对话框或消息框时通知
DisplayingOptionsDialog	single	X		当 Revit 正显示选项对话框时通知
Idling	single	X		当 Revit 不在活动工具或事务中时通知
ViewActivating	pre	X		当 Revit 即将激活某个文件视图时通知
ViewActivated	post	X		在 Revit 已激活某个文件视图后通知

### 5.3.3 注册事件 (Registering Events)

使用此事件是个两步骤的过程。

第一步，必须有一个能处理事件通知的函数。该函数必须接受两个参数，第一个参数对象表示事件通知的“发送者”，第二个参数是事件专有的对象，它含有特定于该事件的事件参数。例如，要注册 `DocumentSavingAs` 事件，事件处理程序必须采用的第二个参数是个 `DocumentSavingAsEventArgs` 对象。

第二步，用 Revit 注册事件。这可以通过 `ControlledApplication` 参数在 `OnStartup()` 函数中尽早实现，或在 Revit 启动后随时实现。虽然事件可以既注册为 `External Command` 又注册为 `External Applications`，但不推荐这样做，除非 `External Command` 是在同一个外部命令中注册和注销该事件。

代码 5-13 演示了注册 `DocumentOpened` 事件，当该事件被触发时，应用程序将设置项目地址。

#### 代码 5-13: 注册 Application.DocumentOpened

```
public class Application_DocumentOpened : IExternalApplication
{
    public IExternalApplication.Result OnStartup (ControlledApplication application)
    {
```



```

        try
        {
            // Register event.
            application.DocumentOpened += new EventHandler
            <Autodesk.Revit.Events.DocumentOpenedEventArgs> (application_DocumentOpened);
        }
        catch (Exception)
        {
            return Autodesk.Revit.UI.Result.Failed;
        }

        return Autodesk.Revit.UI.Result.Succeeded;
    }

    public IExternalApplication.Result OnShutdown (ControlledApplication application)
    {
        // remove the event.
        application.DocumentOpened -= application_DocumentOpened;
        return Autodesk.Revit.UI.Result.Succeeded;
    }

    public void application_DocumentOpened (object sender, DocumentOpenedEventArgs args)
    {
        // get document from event args.
        Document doc = args.Document;

        Transaction transaction = new Transaction (doc, "Edit Address");
        if (transaction.Start() == TransactionStatus.Started)
        {
            doc.ProjectInformation.Address =
                "United States - Massachusetts - Waltham - 1560 Trapelo Road";
            transaction.Commit();
        }
    }
}

```

### 5.3.4 取消事件 (Canceling Events)

操作发生前即已触发的事件 (如“文件保存”) 往往是可取消的 (使用 **Cancellable** 属性确定该事件是否可取消)。例如, 在保存之前可能还想要检查是否满足模型中某些条件。又例如, 注册 **DocumentSaving** 或 **DocumentSavingAs** 事件后, 可以检查文件中的某些条件并取消“保存”或“另存为”操作。一旦“取消”某事件, 则无法“不取消”。

注: 如果“前置事件”被取消, 已订阅该事件的其他事件处理程序不会得到通知。不过, 已订阅与此对应“后置事件”的处理程序会得到通知。

代码 5-14 演示了 **DocumentSavingAs** 事件的事件处理程序 **Project Information Status** 状态参数是否为“空”, 若是, 则取消 **SaveAs** 事件。请注意, 如果您的应用程序取消某事件, 最好给用户一个提示说明。

**代码 5-14: 取消事件**

```
private void CheckProjectStatusInitial (Object sender, DocumentSavingEventArgs args)
{
    Document doc = args.Document;
    ProjectInfo proInfo = doc.ProjectInformation;

    // Project information is only available for project document.
    if (null != proInfo)
    {
        if (string.IsNullOrEmpty (proInfo.Status))
        {
            // cancel the save as process.
            args.Cancel = true;
            MessageBox.Show ("Status project parameter is not set. Save is aborted.");
        }
    }
}
```

注: 虽然大多数事件参数具有 `Cancel` 和 `Cancellable` 属性, 但 `DocumentChanged` 和 `FailuresProcessing` 事件有其相应的 `Cancel()` 和 `IsCancellable()` 方法。

## 5.4 外部事件 (External Events)

Revit API 提供了一个 `External Events` 框架, 以适应使用非模态对话框。它适合于非同步处理, 操作类似于默认频率空闲事件。

要使用 `External Events` 框架来实现非模态对话框, 遵循下列步骤:

- (1) 通过从 `IExternalEventHandler` 接口派生, 实现外部事件处理程序。
- (2) 用静态方法 `ExternalEvent.Create()` 创建 `ExternalEvent`。
- (3) 当非模态对话框发生需要进行 Revit 操作的事件时, 调用 `ExternalEvent.Raise()`。
- (4) 当存在一个可用空闲时间周期时, Revit 将调用 `IExternalEventHandler.Execute()`

方法的实现过程。

### 1. 外部事件处理接口 (`IexternalEventHandler`)

这是外部事件的实现接口。用 Revit 注册实现此接口的类实例, 每次相应外部事件被引发时, 会调用此接口的 `Execute` 方法。

`IExternalEventHandler` 只有两个实现方法, 即 `Execute()` 方法和返回事件名称的 `GetName()` 方法。代码 5-15 演示了一个基本实现过程, 当引发事件时显示一个任务对话框。

**代码 5-15: 实现外部事件处理接口**

```
public class ExternalEventExample : IExternalEventHandler
{
    public void Execute (UIApplication app)
    {
        TaskDialog.Show ("External Event", "Click Close to close.");
    }
}
```



```

public string GetName( )
{
    return "External Event Example";
}
}

```

## 2. 外部事件 ( ExternalEvent )

**ExternalEvent** 类用于创建外部事件。此类的实例会根据事件的创建返回给此外部事件的拥有者。事件的拥有者将使用此实例来标示应由 **Revit** 调用的事件。**Revit** 会周期性检查是否有任何已标示（引发）事件，并通过调用事件各自处理程序的 **Execute** 方法来执行引发的所有事件。

代码 5-16 演示了一个外部应用程序接口的实现过程，它有一个可从 **ExternalCommand** 调用的 **ShowForm()** 方法。**ShowForm()** 方法创建一个外部事件处理程序的新实例，新建一个 **ExternalEvent**，然后显示非模态对话框，最后对话框使用传入的 **ExternalEvent** 对象来引发事件。

### 代码 5-16：创建外部事件

```

public class ExternalEventExampleApp : IExternalApplication
{
    //class instance
    public static ExternalEventExampleApp thisApp = null;
    //ModelessForm instance
    private ExternalEventExampleDialog m_MyForm;

    public Result OnShutdown (UIControlledApplication application)
    {
        if (m_MyForm != null && m_MyForm.Visible)
        {
            m_MyForm.Close( );
        }
        return Result.Succeeded;
    }

    public Result OnStartup (UIControlledApplication application)
    {
        m_MyForm = null;    // no dialog needed yet; the command will bring it
        thisApp = this;    // static access to this application instance
        return Result.Succeeded;
    }

    // The external command invokes this on the end-user's request
    public void ShowForm (UIApplication uiapp)
    {
        // If we do not have a dialog yet, create and show it
        if (m_MyForm == null || m_MyForm.IsDisposed)
        {

```



```
// A new handler to handle request posting by the dialog
ExternalEventExample handler = new ExternalEventExample( );

// External Event for the dialog to use (to post requests)
ExternalEvent exEvent = ExternalEvent.Create ( handler );

// We give the objects to the new dialog;
// The dialog becomes the owner responsible for disposing them, eventually.
m_MyForm = new ExternalEventExampleDialog ( exEvent, handler );
m_MyForm.Show( );
    }
}

[Autodesk.Revit.Attributes.Transaction (Autodesk.Revit.Attributes.TransactionMode.Manual)]
[Autodesk.Revit.Attributes.Regeneration (Autodesk.Revit.Attributes.RegenerationOption.Manual)]
public class Command : IExternalCommand
{
    public virtual Result Execute (ExternalCommandData commandData, ref string message, ElementSet elements)
    {
        try
        {
            ExternalEventExampleApp.thisApp.ShowForm (commandData.Application);
            return Result.Succeeded;
        }
        catch (Exception ex)
        {
            message = ex.Message;
            return Result.Failed;
        }
    }
}
```

### 3. 引发事件 ( Raise Event )

显示非模态对话框后，用户就可在其中交互。对话中的操作可能需要触发某些 Revit 操作。在这种情况下，会调用 `ExternalEvent.Raise()` 方法。代码 5-17 演示了两个按钮的简单非模态对话框代码：一个按钮用于引发事件，一个按钮用于关闭对话框。

#### 代码 5-17：引发事件

```
public partial class ExternalEventExampleDialog : Form
{
    private ExternalEvent m_ExEvent;
    private ExternalEventExample m_Handler;

    public ExternalEventExampleDialog (ExternalEvent exEvent, ExternalEventExample handler)
    {
        InitializeComponent();
        m_ExEvent = exEvent;
        m_Handler = handler;
    }
}
```



```

    }

    protected override void OnFormClosed (FormClosedEventArgs e)
    {
        // we own both the event and the handler
        // we should dispose it before we are closed
        m_ExtEvent.Dispose();
        m_ExtEvent = null;
        m_Handler = null;

        // do not forget to call the base class
        base.OnFormClosed (e);
    }

    private void closeButton_Click (object sender, EventArgs e)
    {
        Close();
    }

    private void showMessageButton_Click (object sender, EventArgs e)
    {
        m_ExtEvent.Raise();
    }
}

```

当 `ExternalEvent.Raise()` 方法被调用时, Revit 将等待一个可用的空闲时间周期, 然后调用 `IExternalEventHandler.Execute()` 方法。在这个简单示例中, 它将显示带有文本 “.” 的任务对话框, 见代码 5-15。

使用 External Events 框架的更复杂例子, 请参阅 SDK 中 `ModelessDialog\ModelessForm_ExternalEvent` 文件夹里的示例代码。该非模态对话框有很多按钮, 且 `IexternalEventHandler` 实现过程有一个公用属性, 用于跟踪哪个按钮被按下, 以便切换 `Execute()` 方法中的值。

## 5.5 可停靠对话框 (Dockable Dialog Panes)

自 Revit 2013 以后的 Revit API 中, 应用程序已经能够利用 `Idling` 事件和 `ExternalEvent` 类来使用非模态对话框。需要非模态对话框的插件也可以选择使用可停靠的非模态对话框。类似于标准的非模态对话框, 可停靠对话框是已注册的、Revit 窗口停靠系统分享的 Windows 图形界面处理 (WPF) 对话框。已注册的可停靠窗格可停靠在 Revit 主窗口内顶部、左侧、右侧和底部, 也可以作为选项卡添加到现有系统窗格中, 如项目浏览器。另外, 可停靠窗格还可以浮动, 与标准的非模态对话框非常相似。

### 1. 可停靠窗格接口 (IDockablePaneProvider)

注册可停靠窗格需要 `IDockablePaneProvider` 接口的一个实例。在 Revit 用户界面初始化期间, 调用此接口的 `SetupDockablePane()` 方法, 以收集有关插件可停靠窗格的窗口信息。`SetupDockablePane()` 有一个 `DockablePaneProviderData` 类型的参数, 这是个关于可停靠窗





格的信息容器。

要实现 `IDockablePaneProvider` 接口, 应设置 `DockablePaneProviderData` 的 `FrameworkElement` 和 `InitialState` 属性。`FrameworkElement` 属性是个包含窗格用户界面的 WPF 对象。

注: 建议插件中的可停靠对话框是实现 `IDockablePaneProvider` 接口的类, 并且从 `System.Windows.Controls.Page` 继承。

`InitialState` 属性是由 `DockablePaneState` 类表示的可停靠窗格初始位置和设置。该窗格的 `DockPosition` 可以为 `Top`、`Bottom`、`Lef`、`Right`、`Floating` 或 `Tabbed`。如果位置是选项卡式的, 则 `DockablePaneState.TabBehind` 属性可用于指定新窗格将显示在哪个窗格之后。如果位置是浮动式的, 则 `DockablePaneState.FloatingRectangle` 属性含有决定窗格大小和位置的矩形。

## 2. 可停靠窗格 (DockablePane)

要在运行时期访问可停靠窗格, 需调用 `UIApplication.RegisterDockablePane()` 方法来注册。该方法需要新窗格的唯一标识符 (`DockablePaneId`)、一个指定窗格标题的字符串, 以及一个 `IDockablePaneProvider` 接口实现。

通过调用 `UIApplication.GetDockablePane()` 并传入唯一 `DockablePaneId` 访问可停靠窗格。此方法返回一个 `DockablePane`。如当前不可见, 则 `DockablePane.Show()` 将在 Revit 用户界面中其最近一次停靠位置显示该窗格。而 `DockablePane.Hide()` 则将可见的可停靠窗格隐藏起来。不过, 这对 Revit 内置的可停靠窗格没有影响。

## 5.6 动态模型更新 (Dynamic Model Update)

动态模型更新为 Revit API 应用程序提供了修改 Revit 模型的能力, 在事务结束即将提交更改时, 对模型中的这些更改作出反应。通过实现 `IUpdater` 接口并用 `UpdaterRegistry` 类将其注册, Revit API 应用程序可以创建更新器。注册包括指定模型中哪些更改会触发更新器。

### 5.6.1 实现更新器接口 (Implementing Iupdater)

`IUpdater` 接口需要实现以下五个方法:

- `GetUpdaterId()`: 此方法会返回更新器的全局唯一标识, 由应用程序 ID 加上此更新器的 GUID 构成。
- `GetUpdaterName()`: 它将返回一个名称, 如果更新器在运行时出现问题, 则用户可据此确定是哪个更新器。
- `GetAdditionalInformation()`: 此方法返回辅助文本, 若更新器未加载, 则 Revit 用此信息通知最终用户。
- `GetChangePriority()`: 此方法标识更新器要执行的变更的性质。用于识别更新器的执行顺序。此方法只在更新器注册期间调用一次。
- `Execute()`: Revit 将调用此方法执行更新。有关 `Execute()` 方法的更多信息, 请参阅 5.6.2 节。



如果某文件被更新器修改, 则该文件将存储更新器的唯一 ID。稍后如果用户打开此文件而更新器不复存在, 则 Revit 将向用户发出警告, 先前编辑文件的第三方更新器不可用, 除非更新器被标记为可选项。默认情况下, 更新器是非可选的, 仅在必要时才使用可选更新器。

代码 5-18 是实现 IUpdater 接口 (更改新建墙的 WallType) 并用 OnStartup() 方法注册更新器的简单示例。它演示了如何创建和使用更新器的所有关键问题。

#### 代码 5-18: 实现更新器接口的示例

```
public class WallUpdaterApplication : Autodesk.Revit.UI.IExternalApplication
{
    public Result OnStartup (Autodesk.Revit.UI.UIControlledApplication application)
    {
        // Register wall updater with Revit
        WallUpdater updater = new WallUpdater (application.ActiveAddInId);
        UpdaterRegistry.RegisterUpdater (updater);

        // Change Scope=any Wall element
        ElementClassFilter wallFilter = new ElementClassFilter (typeof (Wall));

        // Change type=element addition
        UpdaterRegistry.AddTrigger (updater.GetUpdaterId(), wallFilter, Element.GetChangeTypeElementAddition());
        return Result.Succeeded;
    }

    public Result OnShutdown (Autodesk.Revit.UI.UIControlledApplication application)
    {
        WallUpdater updater = new WallUpdater (application.ActiveAddInId);
        UpdaterRegistry.UnregisterUpdater (updater.GetUpdaterId());
        return Result.Succeeded;
    }
}

public class WallUpdater : IUpdater
{
    static AddInId m_appId;
    static UpdaterId m_updaterId;
    WallType m_wallType = null;

    // constructor takes the AddInId for the add-in associated with this updater
    public WallUpdater (AddInId id)
    {
        m_appId = id;
        m_updaterId = new UpdaterId (m_appId, new Guid ("FBFBF6B2-4C06-42d4-97C1-D1B4EB593EFF"));
    }

    public void Execute (UpdaterData data)
    {
        Document doc = data.GetDocument();
```



```
// Cache the wall type
if (m_wallType == null)
{
    FilteredElementCollector collector = new FilteredElementCollector (doc);
    collector.OfClass (typeof (WallType));
    var wallTypes = from element in collector
                    where
                        element.Name == "Exterior - Brick on CMU"
                    select element;
    if (wallTypes.Count<Element>() > 0)
    {
        m_wallType = wallTypes.Cast<WallType>().ElementAt<WallType>(0);
    }
}

if (m_wallType != null)
{
    // Change the wall to the cached wall type.
    foreach (ElementId addedElemId in data.GetAddedElementIds())
    {
        Wall wall = doc.GetElement (addedElemId) as Wall;
        if (wall != null)
        {
            wall.WallType = m_wallType;
        }
    }
}

public string GetAdditionalInformation()
{
    return "Wall type updater example: updates all newly created walls to a special wall";
}

public ChangePriority GetChangePriority()
{
    return ChangePriority.FloorsRoofsStructuralWalls;
}

public UpdaterId GetUpdaterId()
{
    return m_updaterId;
}

public string GetUpdaterName()
{
    return "Wall Type Updater";
}
}
```



### 5.6.2 Execute 方法 (The Execute Method)

Execute()方法的目的是让更新器对文件所做的修改作出反应,并作出适当的关联。Revit 在文件事务结束时调用该方法,该事务中与更新器的 UpdateTrigger 所匹配的图元被添加、更改或删除。同一事务中,此方法可因其他更新程序所做的更改而被多次调用。更新器是在 DocumentChanged 事件之前被调用的,因此该事件将包含所有更新器所作出的更改。

在该方法调用过程中对文件所做的所有更改会成为调用事务的一部分,并为撤销和重做操作保留这些更改。在实现此方法时,将无法打开任何新事务(会引发异常),但如果需要的话,可以使用子事务。

虽然也可以用它来更新文件的外部数据,但这种更改不会成为原始事务的一部分,且当原始事务被撤销或重做时,它不会受到撤销或重做影响。如果使用此方法修改文件的外部数据,则当原始事务被撤销或重做时,还应当订阅 DocumentChanged 事件来更新数据。

#### 1. 更改的范围 (Scope of Changes)

Execute()方法有一个 UpdaterData 参数,提供执行更新所需的全部必要数据,包括触发更新的关于更改的文件和信息。三种基本方法 [GetAddedElementIds()、GetDeletedElementIds()和 GetModifiedElementIds()] 用来识别触发更新的图元。利用 IsChangeTriggered()方法,更新程序还可明确检查是否某个特定更改触发了更新。

#### 2. 更改的禁止和警示 (Forbidden and Cautionary Changes)

当更新器执行时不可调用以下方法,因图元之间引入了交叉引用(当这些更改与工作集结合操作时,可能会导致文件损坏)。当更新器试图调用下列任一方法时,将会引发一个 ForbiddenForDynamicUpdateException:

- Autodesk.Revit.DB.ViewSheet.AddView()。
- Autodesk.Revit.DB.Document.LoadFamily (Autodesk.Revit.DB.Document, Autodesk.Revit.DB.IFamilyLoadOptions)。
- Autodesk.Revit.Creation.Document.NewAreaReinforcement()。
- Autodesk.Revit.Creation.Document.NewPathReinforcement()。

除了上面列出的被禁止方法以外,需要文件处于“事务-空闲”状态的其他 API 方法也不可调用。这些方法包括但不限于: Save()、SaveAs()、Close()、LoadFamily()等,更多信息,请参阅相关方法的说明文档。

UpdaterRegistry 类调用,如 RegistryUpdater()或 AddTrigger(),在更新器 Execute()方法内部亦不允许这些调用。调用任何 UpdaterRegistry 方法都将引发异常。此规则的一个例外是 UpdaterRegistry.UnregisterUpdater()方法,只要未注册的更新器不是当前正在执行的更新器,就可以在更新器执行过程中调用。

虽然在更新器执行期间允许使用以下方法,但若以建立图元之间的交叉参照作为调用结果,则也会引发 ForbiddenForDynamicUpdateException。例如,可以创建与现有的一面墙相交的另一面墙,那么这两面墙会连接在一起。当从更新程序调用以下方法时,应谨慎:



- Autodesk.Revit.Creation.ItemFactoryBase.NewFamilyInstances2()。
- Autodesk.Revit.Creation.ItemFactoryBase.NewFamilyInstance (Autodesk.Revit.DB.XYZ, Autodesk.Revit.DB.FamilySymbol, Autodesk.Revit.DB.Element, Autodesk.Revit.DB.Structure.StructuralType)。
- Autodesk.Revit.Creation.Document.NewFamilyInstance (Autodesk.Revit.DB.XYZ, Autodesk.Revit.DB.FamilySymbol, Autodesk.Revit.DB.Element, Autodesk.Revit.DB.Level, Autodesk.Revit.DB.Structure.StructuralType)。
- Autodesk.Revit.DB.FaceWall.Create()。

还应该指出, 如果修改就能满足要求, 则应避免删除和重新创建现有图元。删除图元可能是个简单的解决方案, 但它不仅会影响 Revit 性能, 而且也会破坏其他图元对“重新创建”对象的任何参照。这可能会导致用户丢失他们已做的正在考虑中的约束和注释图元方面的工作。

### 3. 管理更改 (Managing Changes)

更新器必须能够处理在使用过程中可能引发的复杂问题, 尽可能地满足图元的后续更改。更新器修改过的图元等到下次调用更新器的时候可能又会更改, 而这些更改可能会影响由更新器修改的信息。例如, 图元可被用户显式编辑, 或由重生成触发传播的更改来隐式编辑。

也有可能同一图元被不同的更新器修改, 甚至可能发生在同一事务内部。虽然完全相同的数据显式更改会被跟踪并禁止, 但仍可能发生间接的或传播式的更改。或许最复杂的情况是, 某个图元可能会被用户和/或不同版本文件中的同一更新器更改。在用户重载最新更改或保存到中心文件之后, 会从其他文件带来修改过的目标图元, 并且会需要更新器协调这些更改。

同样重要的是要知道, 当文件与其中心文件进行同步时, 图元的 `ElementId` 可能会受到影响。如果新图元被添加到同一文件的两个版本, 并在两处使用同一 `ElementId`, 则当文件被同步到中心数据库时, 该图元会被协调。因此, 当用更新器交叉参照另一个图元中的某个图元时, 应使用 `Element.UniqueId`, 以保证唯一性。

另一个需要考虑的问题是, 如果更新器将某些数据 (亦即作为参数) 附着给某个图元, 那么它不仅必须确保维持该图元中的信息, 而且, 当该图元是通过复制/粘贴或成组传播来复制的, 这种情况下还得要协调这些数据。例如, 如果更新器通过参数将 “Total weight of rebar” 添加给钢筋的主体图元, 即使钢筋本身可能不随主体一起复制, 该参数及其值也会被拷贝到复制出的钢筋主体图元上。在这种情况下, 需要确保该参数值在新复制的钢筋主体图元中重置。

### 5.6.3 注册更新器 (Registering Updaters)

更新器必须注册以便模型获得更改通知。应用程序级 `UpdaterRegistry` 类提供了注册/注销和操作更新器设置选项的能力。更新器可以由任何 API 回调来注册, 并可注册为应用程序范围级或文件专有级——意味着它们只能由对特定文件所做的更改来触发。为了使用 `UpdaterRegistry` 功能, Revit 插件必须在清单文件中注册, 并且, 对任何更新器 (可由



GetUpdaterId()获得), 由 UpdaterId.GetAddInId()返回的 ID 必须与插件清单文件中的 AddInId 字段匹配。插件无法添加、删除或修改不属于它的更新器。

### 1. 触发器 (Triggers)

除了调用 UpdaterRegistry.RegisterUpdater()方法以外, 更新器还应通过 AddTrigger()方法添加一个或多个更新触发器。这些触发器指示 UpdaterRegistry 什么事件才会触发更新器的 Execute()方法运行。它们可设置为应用程序范围级, 也可设置为只用于特定文件中所做的更改。更新触发器由配对的更改范围和更改类型指定。

更改范围是以下两种之一:

- 文件中图元 ID 的显式列表: 仅这些图元发生更改才会触发更新器。
- 经由 ElementFilter 传出的图元隐式列表: 每个更改过的图元都会在过滤器中运行, 若有任何图元通过, 则触发更新器。

有几个选项可用于更改类型。ChangeTypes 是从 Element 类的静态方法获取的。

- 添加图元: 通过 Element.GetChangeTypeElementAddition()。
- 删除图元: 通过 Element.GetChangeTypeElementDeletion()。
- 更改图元几何 (形状或位置): 通过 Element.GetChangeTypeGeometry()。
- 更改特定参数值: 通过 Element.GetChangeTypeParameter()。
- 图元的任何更改: 通过 Element.GetChangeTypeAny()。

请注意, 触发几何更改的潜在因素很多, 如图元类型的更改、属性和参数的修改、移动、旋转或从其他修改过的图元在重生成期间加于此图元的更改。

还要注意上述最后一个选项, 图元的任何更改, 仅触发对现有图元修改的更新器, 并不触发新添加或删除图元的更新器。此外, 在对实例使用该触发器时, 仅某些类型的修改会触发更新器。影响实例本身的更改, 如实例的几何修改, 会触发更新器。然而, 不直接对实例作修改的更改以及不会导致任何可识别的对实例的更改, 例如更改文本参数, 不会触发此实例的更新器。若要触发基于这些更改的更新, 则其 Type 也必须包括在触发器的更改范围内。

### 2. 执行顺序 (Order of Execution)

为使多个更新器以正确的顺序执行, Revit 需对其排序, 排序主要方式是查看由给定更新器返回的 ChangePriority。报告的优先级为更基本图元集 (如 GridsLevelsReferencePlanes) 的更新器, 将优先于报告优先级为由这些基本图元驱动图元 (如 Annotations) 的更新器执行。为更新器要修改的图元报告适当的更改优先级, 将有益于应用程序的用户, Revit 很少因其他更新器所做的更改而不得不再次执行此更新器。

对于那些有相同更改优先级报告的更新器, 则根据 UpdaterId 排序来顺序执行。方法 UpdaterRegistry.SetExecutionOrder()让您可以在任意两个已注册更新器 (甚至由其他 API 插件注册的更新器) 之间设置执行顺序, 只要你的代码知道这两个更新器的 ID。

## 5.6.4 接触最终用户 (Exposure to End-User)

当更新器正常工作时, 对用户是透明的。但在某些特殊情况下, Revit 会向用户显示一条关于第三方更新器的警告。此类消息将使用 GetUpdaterName()方法的返回值来引用该更



新器。

### 1. 更新器未安装 (Updater not Installed)

如果文件已由某个非可选更新器修改, 后来加载时更新器未安装, 则将显示一个类似图 5-6 所示的任务对话。

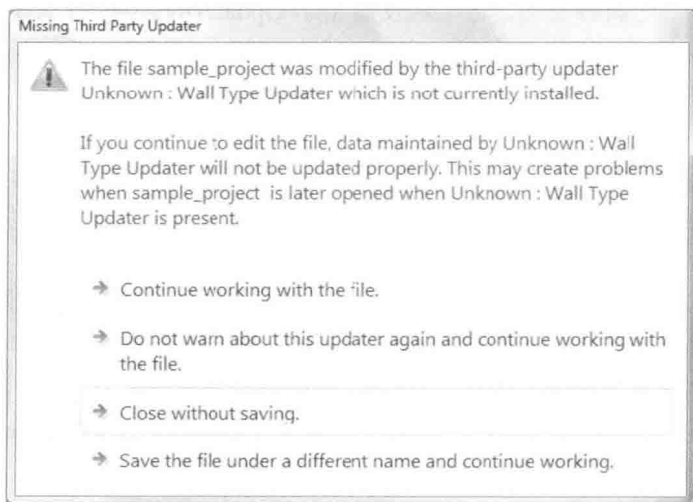


图 5-6 缺少第三方更新器的警告

### 2. 更新器执行无效操作 (Updater Performs Invalid Operation)

若更新器发生错误, 比如一个未处理异常, 则将显示类似图 5-7 的消息, 给用户一个禁用此更新器的选项。



图 5-7 更新器执行无效操作

如果用户选择取消, 则回滚整个事务。例如代码 5-18 墙体更新例子中新添加的墙被删除。如果用户选择禁用更新器, 则不再调用此更新器, 但也不会回滚该事务。

(1) 无限循环 (Infinite Loop)。在更新器陷入无限循环事件时, Revit 将通知用户并禁用此更新器以便 Revit 会话能持续。

(2) 两个更新器试图编辑同一图元 (Two Updaters Attempt to Edit Same Element)。如



果更新器试图编辑同一事务中另一个更新器所更新的图元的同一参数，或者如果试图编辑图元的几何与另一个更新器所做的更改有冲突，则会取消更新器，显示一条错误消息，并给用户一个可以禁用此更新器的选项。

(3) 本地不存在的更新器修改的中心文件 (Central Document Modified by Updater not Present Locally)。如果用户使用本地未安装的更新器修改的中心文件，当重新加载最新更改或保存到中心时，会显示一个任务对话并给出可选择继续或取消同步的选项。警告会提示以后与第三方更新器一起使用时可能引起它与中心模型同步方面的问题。

## 5.7 命令 (Commands)

Revit API 提供了相应的命令可实现现有 Revit 操作界面中位于选项卡、应用程序菜单，或者右键单击菜单所能实现的功能。利用 API 使用 Revit 命令的主要方式有两种，替换现有命令的实现过程或者发布新命令。

### 1. 重写 Revit 命令 (Overriding a Revit command)

AddInCommandBinding 类可用于重写现有 Revit 命令。它有三个替换现有命令实现有关事件。

- **BeforeExecuted:** 此只读事件发生在相关命令执行之前。应用程序可以对此事件作出回应，但不能对文件进行修改，或影响命令的调用。
- **CanExecute:** 当关联的命令启动检查以确定命令是否可在命令目标对象上执行时，发生此事件。
- **Executed:** 当关联命令已执行，且任何重写的实现都已完成时，发生此事件。

要创建命令绑定，调用 UIApplication.CreateAddInCommandBinding() 或 UIControlledApplication.CreateAddInCommandBinding()。两种方法都需要 RevitCommandId id 来标识要替换的命令处理程序。RevitCommandId 有两个获取命令的 ID 静态方法：

- **LookupCommandId:** 用给定的 ID 字符串检索 Revit 命令 ID。为查找命令 ID 字符串，打开一个 Revit 会话，调用所需的命令，关闭 Revit，然后查看该会话的日志。当选取已记录的“Jrn.Command”条目时，其中会有 LookupCommandId() 所需的字符串，如“ID\_EDIT\_DESIGNOPTIONS”。
- **LookupPostableCommandId:** 使用 PostableCommand 枚举检索 Revit 命令 id。这仅适用于可发布的命令（下节中讨论）。

代码 5-19 取自 Revit 2014 SDK 中的 DisableCommand 示例，演示了如何创建 AddInCommandBinding 并重写带有用户消息、禁用原有命令的实现。

### 代码 5-19: 重写命令

```
/// <summary>
/// Implements the Revit add-in interface
/// </summary>
public class Application : IExternalApplication
{
    #region IExternalApplication Members
```





```
/// <summary>
/// Implements the OnStartup event
/// </summary>
/// <param name="application"> </param>
/// <returns> </returns>
public Result OnStartup ( UIControlledApplication application )
{
    // Lookup the desired command by name
    s_commandId = RevitCommandId.LookupCommandId ( s_commandToDisable );

    // Confirm that the command can be overridden
    if ( !s_commandId.CanHaveBinding )
    {
        ShowDialog ( "Error", "The target command " + s_commandToDisable +
            " selected for disabling cannot be overridden" );
        return Result.Failed;
    }

    // Create a binding to override the command.
    // Note that you could also implement .CanExecute to override the accessibility of the command.
    // Doing so would allow the command to be grayed out permanently or selectively, however,
    // no feedback would be available to the user about why the command is grayed out.
    try
    {
        AddInCommandBinding commandBinding = application.CreateAddInCommandBinding ( s_commandId );
        commandBinding.Executed += DisableEvent;
    }
    // Most likely, this is because someone else has bound this command already.
    catch ( Exception )
    {
        ShowDialog ( "Error", "This add-in is unable to disable the target command " + s_commandToDisable +
            "; most likely another add-in has overridden this command." );
    }

    return Result.Succeeded;
}

/// <summary>
/// Implements the OnShutdown event
/// </summary>
/// <param name="application"> </param>
/// <returns> </returns>
public Result OnShutdown ( UIControlledApplication application )
{
    // Remove the command binding on shutdown
    if ( s_commandId.HasBinding )
        application.RemoveAddInCommandBinding ( s_commandId );
    return Result.Succeeded;
}
```



```

}

#endregion

/// <summary>
/// A command execution method which disables any command it is applied to (with a user-visible message) .
/// </summary>
/// <param name="sender">Event sender.</param>
/// <param name="args">Arguments.</param>
private void DisableEvent (object sender, ExecutedEventArgs args)
{
    ShowDialog ( "Disabled", "Use of this command has been disabled.");
}

/// <summary>
/// Show a task dialog with a message and title.
/// </summary>
/// <param name="title">The title.</param>
/// <param name="message">The message.</param>
private static void ShowDialog (string title, string message)
{
    // Show the user a message.
    TaskDialog td = new TaskDialog (title)
    {
        MainInstruction = message,
        TitleAutoPrefix = false
    };
    td.Show();
}

/// <summary>
/// The string name of the command to disable. To lookup a command id string, open a session of Revit,
/// invoke the desired command, close Revit, then look to the journal from that session. The command
/// id string will be toward the end of the journal, look for the "Jrn.Command" entry that was recorded
/// when it was selected.
/// </summary>
static String s_commandToDisable = "ID_EDIT_DESIGNOPTIONS";

/// <summary>
/// The command id, stored statically to allow for removal of the command binding.
/// </summary>
static RevitCommandId s_commandId;
}

```

## 2. 发布命令 ( Posting a Command )

当控制从当前的 API 应用程序返回时, 方法 `UIApplication.PostCommand()` 将发布一个命令到被调用的 Revit 消息队列。只有某些命令可以这种方式发布。它们包括 `Autodesk.Revit.`



UI.PostableCommand 枚举类型中的所有命令以及由插件创建的任何外部命令。

注：当使用 PostCommand() 时，有可能发布的命令无法执行。一个可能的原因是已经发布了另一个命令。在给定的时间只能有一个命令发布到 Revit，因此如果发布第二个命令，则 PostCommand() 将引发异常。发布命令无法执行的另一个可能的原因是，在命令不可访问时要执行该命令。它是否可访问仅取决于 Revit 从 API 上下文环境中何处返回，因而由此产生的执行失败不会直接报告给发布命令的应用程序。

UIApplication.CanPostCommand() 用来识别给定的命令是否可发布，这意味着它是 PostableCommand 的成员还是外部命令。但它并不识别该命令当前是否可访问。

PostCommand() 和 CanPostCommand() 两者都需要一个 RevitCommandId，它可以像代码 5-19 所述方式获取。

## 5.8 故障发布和处理 (Failure Posting and Handling)

当某个用户可见的问题发生时，Revit API 可发布该故障，并能对 Revit 或 Revit 插件发布的故障作出响应。

### 5.8.1 发布故障 (Posting Failures)

要使用故障发布机制来报告问题，请遵循以下步骤：

(1) 在外部应用程序的 OnStartup() 调用期间，Revit 尚未定义的新故障必须在 FailureDefinitionRegistry 中定义和注册。

(2) 可以从 BuiltInFailures 类或者从使用 FailureDefinition 类相关的预注册的自定义故障中，找出故障定义 ID。

(3) 使用 FailureMessage 相关的类设置故障相关的选项和详细信息，发布有问题文件的故障。

#### 1. 故障定义和注册 (Defining and Registering a Failure)

在应用程序启动期间，通过创建 FailureDefinition 对象，每个可能的 Revit 故障都必须定义并注册。FailureDefinition 对象包含有关故障的一些永久性信息，如特征、严重程度、基本描述、解决方案的类型及默认的解决方案。

代码 5-20 演示了如何创建两个新故障，可用于墙太高时发出“警告”和“错误”。这个例子中，它们与执行故障发布的 Updater 结合使用。FailureDefinitionIds 保存于 Updater 类中，因为发布故障时需要用到它们。下节将会更详细地解释 FailureDefinition.CreateFailureDefinition() 方法参数。

#### 代码 5-20：定义并注册故障

```
WallWarnUpdater wallUpdater = new WallWarnUpdater();
UpdaterRegistry.RegisterUpdater (wallUpdater);
ElementClassFilter filter = new ElementClassFilter (typeof (Wall));
UpdaterRegistry.AddTrigger (wallUpdater.GetUpdaterId(), filter, Element.GetChangeTypeGeometry());

// define a new failure id for a warning about walls
```



```
FailureDefinitionId warnId = new FailureDefinitionId (new Guid ("FB4F5AF3-42BB-4371-B559-FB1648D5B4D1"));

// register the new warning using FailureDefinition
FailureDefinition failDef = FailureDefinition.CreateFailureDefinition (warnId, FailureSeverity.Warning, "Wall is too big (>100') . Performance problems may result.");

FailureDefinitionId failId = new FailureDefinitionId (new Guid ("691E5825-93DC-4f5c-9290-8072A4B631BC"));

FailureDefinition failDefError = FailureDefinition.CreateFailureDefinition (failId, FailureSeverity.Error, "Wall is WAY too big (>200') . Performance problems may result.");

// save ids for later reference
wallUpdater.WarnId = warnId;
wallUpdater.FailureId = failId;
```

### 2. 故障定义 ID (FailureDefinitionId)

作为注册故障定义的关键, 必须使用唯一 FailureDefinitionId。每个唯一 FailureDefinitionId 都应使用 GUID 生成工具创建。然后, 该故障定义 ID 可以用于在 FailureDefinitionRegistry 中查找 FailureDefinition, 以及创建和发布 FailureMessages。

### 3. 严重程度 (Severity)

注册新的故障时, 随 FailureDefinitionId 和用户可见的故障文本描述一起, 指定其严重程度。严重程度确定在文件中允许哪些操作以及是否可以提交该事务。严重程度选项有:

- **Warning:** 最终用户可以忽略的故障。这一严重程度的故障不会阻止事务被提交。在 Revit 需要将问题传递给用户, 但问题并不妨碍用户继续在文件上操作的情况下, 应使用这一严重程度选项。
- **Error:** 不可忽略的故障。如果发布这种严重程度的 FailureMessage, 除非已通过适当的 FailureResolution 解决了故障, 否则无法提交当前事务。在文件操作不能继续时, 除非问题已得到解决, 否则应使用此严重程度选项。如果该故障没有预定义的解决方案可用, 或这些解决方案未能解决问题, 则必须中止事务以便继续使用该文件。强烈建议, 每个这种严重程度的故障至少有一个解决方案。
- **DocumentCorruption:** 由已知的文件损坏所造成的故障。此故障会强制 Transaction 尽快回滚。当发布这种严重程度的故障时, 不允许从文件中读取信息。必须首先回滚当前事务以便继续处理文件。这种严重程度选项仅用于假若有已知的文件数据损坏的情况。除非是没办法预防损坏或从本地恢复, 这种类型的故障通常应该避免使用。

当定义新 FailureDefinition 时, 严重程度除了以上三种选项, 无其他选项。

### 4. 故障解决方案 (Failure Resolutions)

在故障可被解决的情况下, 所有可能的解决方案都应该在 FailureDefinition 类中预先定义。它可通知 Revit 哪个故障解决方案可用于给定的故障。FailureDefinition 包含了适用于故障处理的完整的解决方案类型列表, 包括用户可见的解决方案标题。

解决方案数量没有限制, 但是直到 2011 Revit API 为止, 公开的故障解决方案仅仅是 DeleteElements。当指定多个解决方案时, 除非使用 SetDefaultResolutionType() 方法显式更



改, 所添加的第一个解决方案将成为默认解决方案。如果适用的话, Revit 故障处理机制会自动使用默认解决方案来解决故障。Revit UI 只使用默认解决方案, 但是通过 Revit API, Revit 插件可使用任何适用的解决方案, 并能提供一个可供替代的 UI 来做这些 (如 5.8.2 节所述)。

在故障严重程度为 `DocumentCorruption` 的情况下, 等到故障解决方案可以处理时, 事务已经终止, 已没什么问题要解决。因此, 严重程度为 `DocumentCorruption` 的 API 定义故障不需要添加 `FailureResolutions`。

### 5. 发布故障 (Posting a Failure)

`Document.PostFailure()` 方法用于通知问题文件。故障会得到确认并可能在事务结束时得到解决。通过此方法发布的警告在得到解决后不再存储在文件中。故障发布用于指出在事务结束前可能改变的文件状态, 或者在有理由将解决方案推迟到事务结束的情况下使用。并非外部命令遇到的所有故障都必须发布, 如果故障与文件无关, 则应使用任务对话。例如, 假若 Revit UI 处于执行外部命令无效状态。

要发布故障, 在自定义故障被定义后, 就可用 `FailureDefinitionId` 创建新 `FailureMessage`, 或使用 Revit API 提供的 `BuiltInFailure`。在 `FailureMessage` 对象中设置附加信息, 如故障图元, 然后调用 `Document.PostFailure()` 传入新建的 `FailureMessage`。请注意, 文件必须为可修改的, 以便发布故障。

由 `PostFailure()` 返回的唯一 `FailureMessageKey` 可在事务的生存期内存储, 并可用于删除不再有意义的故障消息。若多次发布同一故障消息, 则返回的 `FailureMessageKey` 也相同。若发布的故障为“文件损坏”级严重程度, 则返回一个无效 `FailureMessageKey`。这是因为文件损坏故障不能“消除发布”。

代码 5-21 演示了根据 `Execute()` 方法所收到信息来发布故障的 `IUpdater` 类 (引用代码 5-20 “定义并注册故障”代码)。

#### 代码 5-21: 发布故障

```
public class WallWarnUpdater : IUpdater
{
    static AddInId m_appld;
    UpdaterId m_updaterId;
    FailureDefinitionId m_failureId = null;
    FailureDefinitionId m_warnId = null;

    // constructor takes the AddInId for the add-in associated with this updater
    public WallWarnUpdater (AddInId id)
    {
        m_appld = id;
        m_updaterId = new UpdaterId (m_appld, new Guid ("69797663-7BCB-44f9-B756-E4189FE0DED8"));
    }

    public void Execute (UpdaterData data)
    {
        Document doc = data.GetDocument();
```



```

Autodesk.Revit.ApplicationServices.Application app = doc.Application;
foreach (ElementId id in data.GetModifiedElementIds())
{
    Wall wall = doc.GetElement (id) as Wall;
    Autodesk.Revit.DB.Parameter p = wall.get_Parameter ("Unconnected Height");
    if (p != null)
    {
        if (p.AsDouble() > 200)
        {
            FailureMessage failMessage = new FailureMessage (FailureId);
            failMessage.SetFailingElement (id);
            doc.PostFailure (failMessage);
        }
        else if (p.AsDouble() > 100)
        {
            FailureMessage failMessage = new FailureMessage (WarnId);
            failMessage.SetFailingElement (id);
            doc.PostFailure (failMessage);
        }
    }
}

public FailureDefinitionId FailureId
{
    get { return m_failureId; }
    set { m_failureId = value; }
}

public FailureDefinitionId WarnId
{
    get { return m_warnId; }
    set { m_warnId = value; }
}

public string GetAdditionalInformation()
{
    return "Give warning and error if wall is too tall";
}

public ChangePriority GetChangePriority()
{
    return ChangePriority.FloorsRoofsStructuralWalls;
}

public UpdaterId GetUpdaterId()
{
    return m_updaterId;
}

```



```
public string GetUpdaterName()  
{  
    return "Wall Height Check";  
}
```

#### 6. 消除已发布故障 (Removal of Posted Failures)

同一事务中，文件可能有多次更改、多次重生成，因此很可能有些故障不再有意义，需要删除以防“假警报”。通过调用 `Document.UnpostFailure()`，并传入 `PostFailure()`调用获得 `FailureMessageKey`，可消除已发布的特定消息。若故障的严重程度是 `Document.Corruption`，则 `UnpostFailure()`将引发异常。

当事务即将发生回滚时，通过使用 `Transaction.SetFailureHandlingOptions()`方法，也可以自动消除所有已发布的故障（避免用户单击**取消**按钮）。

### 5.8.2 处理故障 (Handling Failures)

通常，已发布的故障由 Revit 的标准故障解决方案 UI 在事务结束时处理 [具体来说是指 `Transaction.Commit()`或 `Transaction.Rollback()`被调用时]，并向用户呈现处理故障的信息和选项。

如果文件的某项操作（或一系列操作）需要从 Revit 插件中对某些错误作一些特殊处理，则可定制故障处理以实现这一解决方案。可提供的自定义故障处理如下：

- 对给定的事务，使用接口 `IFailuresPreprocessor`。
- 对所有可能的错误，使用 `FailuresProcessing` 事件。

最后，利用接口 `IFailuresProcessor`，API 提供了可完全替代标准的故障处理用户界面的能力。虽然在大多数情况下，前两个方法处理故障应已足够，最后的选项可用于一些特殊场合，如提供一个更好的故障处理 UI，或当应用程序用作为 Revit 顶层的前端程序。

#### 1. 故障处理概述 (Overview of Failure Processing)

重要的是要记住在调用 `Transaction.Commit()`和实际处理故障之间会发生很多事情。自动连接、重叠检查、成组检查和工作集可编辑性检查，仅是略举几例。这些检查和更改可能会使一些故障不复存在，或者更有可能发布新故障。因此，当调用 `Transaction.Commit()`时对要处理的故障状态还不能得出结论。要正确处理故障，需挂接到实际故障处理机制。

在故障处理开始时，事务中对文件该进行的所有更改都已完成，所有故障都已发布。因此，在故障处理期间不允许有不受控的文件更改。通过 `FailuresAccessor` 提供的受限接口，可得到有限的解决故障能力。如果这已发生，则所有事务结束检查和故障处理都必须重做。因此在一个事务末，可能会有几次故障解决方案循环。

故障处理的每个循环包括三个步骤：

- (1) 故障预处理 (`FailuresPreprocessor`)。
- (2) 故障处理事件广播 (`FailuresProcessing` 事件)。
- (3) 最终处理 (`FailuresProcessor`)。

根据返回的不同 `FailureProcessingResults`，这三个步骤的每一步都可以控制接下来会发



生什么。选项有:

- **Continue**: 对执行流没有影响。如果 `FailuresProcessor` 返回带有未解决故障的“Continue”, Revit 会表现的更像返回了“`ProceedWithRollBack`”。
- **ProceedWithCommit**: 中断当前故障处理并伴随着另一个故障处理立即触发另一个事务末检查循环。在尝试解决该故障后应当返回。如果返回没有任何成功的故障解决方案,则很容易导致无限循环。如果事务已回滚则无法返回,这种情况会被视为“`ProceedWithRollBack`”。
- **ProceedWithRollback**: 继续执行故障处理,但会强制事务回滚,即使它原本请求提交。如果在 `ProceedWithRollBack` 返回之前, `FailureHandlingOptions` 已设置为回滚后清除错误,则不会发生进一步的错误处理,所有故障都会被删除而静静地回滚事务。否则,默认故障处理将继续,故障可能被提交给用户,但能确保事务被回滚。
- **WaitForUserInput**: 如果它正等待外部事件(通常是用户输入)来完成故障处理,那么只能通过 `FailuresProcessor` 返回。

根据事务中所发布故障的严重程度以及事务是被提交还是回滚,这三个步骤的每一步都可能几个解决错误的选项。所有关于文件中已发布故障的信息、为解决故障而执行某些操作的能力以及执行这些操作的 API 信息,都是通过 `FailuresAccessor` 类来提供。`Document` 类可用来获取附加信息,但不同于 `FailuresAccessor` 类, `Document` 类提供的附加信息不能更改。

## 2. 故障访问器 (`FailuresAccessor`)

`FailuresAccessor` 对象作为参数传递到故障处理的每个步骤,并且是获取文件中有关故障信息的唯一接口。当在故障处理期间允许从文件中读取时,通过使用此类所提供的方法,是故障解决期间修改文件的唯一途径。在故障处理返回后,该类的实例即失效,不再可使用。

(1) 故障访问器的可用信息 (Information Available from `FailuresAccessor`)。 `FailuresAccessor` 对象提供一些通用信息,如:

- 故障正在处理或预处理的文件。
- 文件中所发布故障的最高严重程度。
- 事务名称和完成事务的故障处理选项。
- 要求事务提交还是回滚。

通过 `GetFailuresMessages()` 方法, `FailuresAccessor` 对象还提供了特定故障信息。

(2) 故障解决选项 (Options to resolve failures)。 `FailuresAccessor` 对象提供以下方法来解决故障:

- `DeleteWarning()` 或 `DeleteAllWarnings()` 方法,可删除严重程度为 `Warning` 的故障消息。
- `ResolveFailure()` 或 `ResolveFailures()` 方法,使用每个故障最后的故障解决方案类型设置,解决一个或多个故障。
- `DeleteElements()`, 通过删除故障相关图元来解决故障。
- 或者用 `ReplaceFailures()` 方法,删除所有故障消息并代之以一个“generic”故障。





### 3. 故障预处理器接口 (IFailuresPreprocessor)

IFailuresPreprocessor 仅可用于为特定事务提供自定义故障处理。IFailuresPreprocessor 是一个接口, 用于执行预处理步骤以滤出预期的事务故障或发布新故障 (代码 5-22)。故障可通过以下方式被“滤出”。

- 删除事务发布的已知警告, 和特定事务上下文中被视为与用户无关的警告。
- 解决事务发布的某些已知故障, 以及应在给定事务上下文中解决的某些故障。
- 在不应提交“有缺陷”事务的情况下, 或对于某个给定工作流, 中止事务比用户交互更可取的情况下, 中止事务。

IFailuresPreprocessor 接口在故障解决过程期间首先获取控制。IFailuresPreprocessor 在确保所有故障发布之后和/或所有不相关的故障被删除之后, 在适当时间获得控制, 除此而外, 几乎相当于在完成事务之前检查和解决故障。

每个事务可能只有一个 IFailuresPreprocessor, 且没有默认的故障预处理器。如果没有附加到事务的故障预处理方案 (通过故障处理选项), 故障解决方案的第一步则被简单忽略。

#### 代码 5-22: 从 IFailuresPreprocessor 处理故障

```
public class SwallowTransactionWarning : IExternalCommand
{
    public Autodesk.Revit.UI.Result Execute (ExternalCommandData commandData, ref string message, ElementSet elements)
    {
        Autodesk.Revit.ApplicationServices.Application app =
            commandData.Application.Application;
        Document doc = commandData.Application.ActiveUIDocument.Document;
        UIDocument uidoc = commandData.Application.ActiveUIDocument;

        FilteredElementCollector collector = new FilteredElementCollector (doc);
        ICollection<Element> elementCollection =
            collector.OfClass (typeof (Level)).ToElements();
        Level level = elementCollection.Cast<Level>().ElementAt<Level> (0);

        Transaction t = new Transaction (doc);
        t.Start ("room");
        FailureHandlingOptions failOpt = t.GetFailureHandlingOptions();
        failOpt.SetFailuresPreprocessor (new RoomWarningSwallower());
        t.SetFailureHandlingOptions (failOpt);

        doc.Create.NewRoom (level, new UV (0, 0));
        t.Commit();

        return Autodesk.Revit.UI.Result.Succeeded;
    }
}

public class RoomWarningSwallower : IFailuresPreprocessor
{
    public FailureProcessingResult PreprocessFailures (FailuresAccessor failuresAccessor)
    {
    }
```



```

IList<FailureMessageAccessor> failList = new List<FailureMessageAccessor>( );
// Inside event handler, get all warnings
failList = failuresAccessor.GetFailureMessages( );
foreach (FailureMessageAccessor failure in failList)
{
    // check FailureDefinitionIds against ones that you want to dismiss , FailureDefinitionId failID =
    failure.GetFailureDefinitionId( );
    // prevent Revit from showing Unenclosed room warnings
    if ( failID == BuiltInFailures.RoomFailures.RoomNotEnclosed )
    {
        failuresAccessor.DeleteWarning ( failure );
    }
}

return FailureProcessingResult.Continue;
}
}

```

#### 4. 故障处理事件 (FailuresProcessing Event)

FailuresProcessing 事件最适合那些想提供无用户界面的自定义故障处理的应用程序，无论是对整个会话还是对许多不相关的事务。通过此事件处理故障的一些用法如下：

- 自动去除某些警告和/或自动解决基于办公标准（或其他标准）的某些错误。
- 自定义故障记录。

FailuresProcessing 事件会在 IFailuresPreprocessor（如果有的话）完成后引发。它可以有任意数量的处理程序，且都将会被调用。由于事件处理程序无法返回“值”，因此必须用带事件参数的 SetProcessingResult() 来传递状态。参数仅可设置为 Continue、Proceed WithRollback 或 ProceedWithCommit。

代码 5-23 演示了 FailuresProcessing 事件的故障处理。

#### 代码 5-23：处理故障处理事件

```

private void CheckWarnings (object sender, FailuresProcessingEventArgs e)
{
    FailuresAccessor fa = e.GetFailuresAccessor( );
    IList<FailureMessageAccessor> failList = new List<FailureMessageAccessor>( );
    failList = fa.GetFailureMessages( ); // Inside event handler, get all warnings
    foreach (FailureMessageAccessor failure in failList)
    {
        // check FailureDefinitionIds against ones that you want to dismiss , FailureDefinitionId failID =
        failure.GetFailureDefinitionId( );
        // prevent Revit from showing Unenclosed room warnings
        if ( failID == BuiltInFailures.RoomFailures.RoomNotEnclosed )
        {
            fa.DeleteWarning ( failure );
        }
    }
}
}

```



### 5. 故障处理器 (FailuresProcessor)

在 FailuresProcessing 事件处理之后, IFailuresProcessor 接口获得最终控制。Revit 会话中只能有一个活动的 IFailuresProcessor。要注册故障处理器, 请使用 Application.RegisterFailuresProcessor() 方法从 IFailuresProcessor 派生一个类并注册。如果已有先前注册的故障处理器, 则它被丢弃。如果 Revit 插件选择为 Revit 注册一个故障处理器, 则该处理器将成为所有 Revit 会话错误的默认错误处理程序, 而标准的 Revit 错误对话框则不会出现。若未设置故障处理器, 则 Revit UI 中有个可调用所有常规 Revit 错误对话框的默认故障处理器。重写带有自定义故障解决方案处理程序的 FailuresProcessor, 它可以是交互式的, 也可以没有用户界面, 仅用于取代现有的 Revit 故障 UI。

如果 RegisterFailuresProcessor() 方法传递 null, 有任何故障的任何事务, 都将无声地中止 (除非故障已在故障处理的前两个步骤中解决)。

IFailuresProcessor.ProcessFailures() 方法允许返回 WaitForUserInput, 使事务挂起。在这种情况下, 预计 FailuresProcessor 会在屏幕上留下某一用户界面, 并最终将提交或回滚此前挂起的事务, 否则挂起状态将无限期持续下去, 相当于文件被冻结。

代码 5-24 使用 IFailuresProcessor 检查故障, 删除故障图元并为用户设置相应的消息。

代码 5-24: 故障处理器接口

```
[Autodesk.Revit.Attributes.Transaction(Autodesk.Revit.Attributes.TransactionMode.Automatic)] public class MyFailuresUI :
IExternalApplication
{
    static AddInId m_appId = new AddInId (new Guid ("9F179363-B349-4541-823F-A2DDB2B86AF3"));
    public Autodesk.Revit.UI.Result OnStartup (UIControlledApplication uiControlledApplication)
    {
        IFailuresProcessor myFailUI = new FailUI();
        Autodesk.Revit.ApplicationServices.Application.RegisterFailuresProcessor (myFailUI);
        return Result.Succeeded;
    }

    public Autodesk.Revit.UI.Result OnShutdown (UIControlledApplication application)
    {
        return Result.Succeeded;
    }

    public class FailUI : IFailuresProcessor
    {
        public void Dismiss (Document document)
        {
            // This method is being called in case of exception or document destruction to
            // dismiss any possible pending failure UI that may have left on the screen
        }

        public FailureProcessingResult ProcessFailures (FailuresAccessor failuresAccessor)
        {
            IList<FailureResolutionType> resolutionTypeList =
                new List<FailureResolutionType>();
        }
    }
}
```



```

IList<FailureMessageAccessor> failList = new List<FailureMessageAccessor>( );
// Inside event handler, get all warnings
failList = failuresAccessor.GetFailureMessages( );
string errorString = "";
bool hasFailures = false;
foreach (FailureMessageAccessor failure in failList)
{
    // check how many resolutions types were attempted to try to prevent
    // entering infinite loop
    resolutionTypeList =
        failuresAccessor.GetAttemptedResolutionTypes (failure);
    if (resolutionTypeList.Count >= 3)
    {
        TaskDialog.Show ("Error", "Cannot resolve failures - transaction will be rolled back.");
        return FailureProcessingResult.ProceedWithRollBack;
    }

    errorString += "IDs ";
    foreach (ElementId id in failure.GetFailingElementIds( ))
    {
        errorString += id + ", ";
        hasFailures = true;
    }
    errorString += "\nWill be deleted because: " + failure.GetDescriptionText( ) + "\n";
    failuresAccessor.DeleteElements (
        failure.GetFailingElementIds( ) as IList<ElementId>);
}
if (hasFailures)
{
    TaskDialog.Show ("Error", errorString);
    return FailureProcessingResult.ProceedWithCommit;
}

return FailureProcessingResult.Continue;
}
}
}

```

## 5.9 性能顾问 (Performance Adviser)

Revit API 性能顾问功能旨在分析文件并为用户标记任何可能导致性能下降的图元和/或设置。Performance Adviser 命令执行一组规则并将其结果显示在一个标准的审查警告对话框中。

API 中有两个有关性能顾问的类:

- **PerformanceAdviser**: 一个具有双重作用的应用程序范围的对象, 其一是作为检测潜在性能问题的运行规则的注册, 其二是作为执行这些规则的引擎。



- **IPerformanceAdviserRule**: 一个接口, 可定义 Performance Adviser 的新规则。

### 1. 性能顾问 (Performance Adviser)

**PerformanceAdviser** 用来添加或删除检查用的规则、启用和禁用规则, 获取列表中的规则信息, 执行列表中的某些或所有规则。应用程序一般在程序启动期间使用 **AddRule()** 创建新规则, 在应用程序关闭期间使用 **DeleteRule()** 注销它。**ExecuteAllRules()** 将执行给定文件列表中的所有规则, 而 **ExecuteRules()** 用于执行文件中所选定的规则。这两种方法都会返回一个故障消息列表, 解释文件中检测到的性能问题。

代码 5-25 演示了如何遍历所有性能顾问规则, 并对文件执行所有这些规则。

#### 代码 5-25: 性能顾问

```
//Get the name of each registered PerformanceRule and then execute all of them.
foreach (PerformanceAdviserRuleId id in PerformanceAdviser.GetPerformanceAdviser().GetAllRuleIds())
{
    string ruleName = PerformanceAdviser.GetPerformanceAdviser().GetRuleName(id);
}
PerformanceAdviser.GetPerformanceAdviser().ExecuteAllRules(document);
```

### 2. 性能顾问规则接口 (IPerformanceAdviserRule)

创建 **IPerformanceAdviserRule** 接口实例来创建 Performance Adviser 的新规则。这些规则可特定于图元, 也可以是文件范围的规则。需要实现以下方法:

- **GetName()**: 命名规则用的简短字符串。
- **GetDescription()**: 一两个规则说明语句。
- **InitCheck()**: 性能顾问启动检查时所调用的方法。如果规则是检查整个文件, 而不是具体的图元, 应以此方法进行检查。
- **FinalizeCheck()**: 性能顾问结束检查时所调用的方法。在规则执行期间发现的任何问题的结果, 可以在这期间使用 **FailureMessage(s)** 来报告。
- **WillCheckElements()**: 指示是否要对个别的图元执行规则。
- **GetElementFilter()**: 检索过滤器以限制所要检查的图元。
- **ExecuteElementCheck()**: 性能顾问对每个需要检查的图元所调用的方法。

代码 5-26 演示了如何实现自定义规则, 以识别文件中任何有翻转饰面的门, 摘录于 Revit API SDK Samples 文件夹中的“PerformanceAdviserControl”示例(请参阅该示例项目, 以得到完整的类实现)。

#### 代码 5-26: 实现 IPerformanceAdviserRule

```
public class FlippedDoorCheck : Autodesk.Revit.DB.IPerformanceAdviserRule
{
    #region Constructor
    /// <summary>
    /// Set up rule name, description, and error handling
    /// </summary>
    public FlippedDoorCheck()
    {
        m_name = "Flipped Door Check";
    }
}
```



```

        m_description = "An API-based rule to search for and return any doors that are face-flipped";
        m_doorWarningId = new Autodesk.Revit.DB.FailureDefinitionId (new Guid ("25570B8FD4AD42baBD78469ED60FB9A3"));
        m_doorWarning = Autodesk.Revit.DB.FailureDefinition.CreateFailureDefinition ( m_doorWarningId, Autodesk.Revit.DB.
FailureSeverity.Warning, "Some doors in this project are face-flipped.");
    }
    #endregion

    #region IPerformanceAdviserRule implementation
    /// <summary>
    /// Does some preliminary work before executing tests on elements. In this case,
    /// we instantiate a list of FamilyInstances representing all doors that are flipped.
    /// </summary>
    /// <param name="document">The document being checked</param>
    public void InitCheck (Autodesk.Revit.DB.Document document)
    {
        if (m_FlippedDoors == null)
            m_FlippedDoors = new List<Autodesk.Revit.DB.ElementId>( );
        else
            m_FlippedDoors.Clear( );
        return;
    }

    /// <summary>
    /// This method does most of the work of the IPerformanceAdviserRule implementation.
    /// It is called by PerformanceAdviser.
    /// It examines the element passed to it (which was previously filtered by the filter
    /// returned by GetElementFilter() (see below)). After checking to make sure that the
    /// element is an instance, it checks the FacingFlipped property of the element.
    ///
    /// If it is flipped, it adds the instance to a list to be used later.
    /// </summary>
    /// <param name="document">The active document</param>
    /// <param name="element">The current element being checked</param>
    public void ExecuteElementCheck (Autodesk.Revit.DB.Document document, Autodesk.Revit.DB.Element element)
    {
        if ((element is Autodesk.Revit.DB.FamilyInstance))
        {
            Autodesk.Revit.DB.FamilyInstance doorCurrent = element as Autodesk.Revit.DB.FamilyInstance;
            if (doorCurrent.FacingFlipped)
                m_FlippedDoors.Add (doorCurrent.Id);
        }
    }

    /// <summary>
    /// This method is called by PerformanceAdviser after all elements in document
    /// matching the ElementFilter from GetElementFilter() are checked by ExecuteElementCheck().
    ///
    ///

```



```
/// This method checks to see if there are any elements (door instances, in this case) in the
/// m_FlippedDoor instance member. If there are, it iterates through that list and displays
/// the instance name and door tag of each item.
/// </summary>
/// <param name="document">The active document</param>
public void FinalizeCheck (Autodesk.Revit.DB.Document document)
{
    if (m_FlippedDoors.Count == 0)
        System.Diagnostics.Debug.WriteLine ("No doors were flipped. Test passed.");
    else
    {
        //Pass the element IDs of the flipped doors to the revit failure reporting APIs.
        Autodesk.Revit.DB.FailureMessage fm = new Autodesk.Revit.DB.FailureMessage (m_doorWarningId);
        fm.SetFailingElements (m_FlippedDoors);
        Autodesk.Revit.DB.Transaction failureReportingTransaction = new Autodesk.Revit.DB.Transaction (document,
"Failure reporting transaction");
        failureReportingTransaction.Start( );
        document.PostFailure (fm);
        failureReportingTransaction.Commit( );
        m_FlippedDoors.Clear( );
    }
}

/// <summary>
/// Gets the description of the rule
/// </summary>
/// <returns>The rule description</returns>
public string GetDescription( )
{
    return m_description;
}

/// <summary>
/// This method supplies an element filter to reduce the number of elements that PerformanceAdviser
/// will pass to GetElementCheck( ). In this case, we are filtering for door elements.
/// </summary>
/// <param name="document">The document being checked</param>
/// <returns>A door element filter</returns>
public Autodesk.Revit.DB.ElementFilter GetElementFilter (Autodesk.Revit.DB.Document document)
{
    return new Autodesk.Revit.DB.ElementCategoryFilter (Autodesk.Revit.DB.BuiltInCategory.OST_Doors);
}

/// <summary>
/// Gets the name of the rule
/// </summary>
/// <returns>The rule name</returns>
public string GetName( )
{

```



```

        return m_name;
    }
    /// <summary>
    /// Returns true if this rule will iterate through elements and check them, false otherwise
    /// </summary>
    /// <returns>True</returns>
    public bool WillCheckElements( )
    {
        return true;
    }
    #endregion
}

```

## 5.10 点云 (Point Clouds)

Revit API 提供了两种使用点云的方法。第一种方法允许创建新的点云实例，读取并过滤点、选择全部点的子集、选择高亮显示或个别的点。第二种方法允许使用自己的云引擎并处理非支持的文件格式（亦即.pcg, .rcp 或.rcs 以外的格式），向 Revit 提供用户可以看到点。

客户端 API:

- 创建新点云实例。
- 读取并过滤点数据。
- 点集选择。
- 控制点云高亮显示。

引擎 API:

- 注册点云文件扩展名。
- 向 Revit 提供要渲染的点。

### 5.10.1 点云客户端 (Point Cloud Client)

点云客户端 API 支持在 Revit 中读取和修改点云实例。点云实例提供的点来自点云引擎，可以是 Revit 的内置引擎，也可以是作为应用程序加载的第三方引擎。客户端点云 API 应用程序无需关心引擎如何存储并向 Revit 提供点的细节。而事实上，客户端 API 可用于创建点云、操控其属性、读取与给定过滤器匹配所找出的点。

点云相关的主类有:

- **PointCloudType**: 加载到 Revit 文件的点云类型。每个 PointCloudType 映射到单个文件或标识符（取决于控制它的点云引擎类型）。
- **PointCloudInstance**: Revit 项目中某个位置的点云实例。
- **PointCloudFilter**: 过滤器，用来确定提取哪些点。
- **PointCollection**: 从实例和过滤器中获取的点集合。
- **PointIterator**: PointCollection 中的点迭代器。
- **CloudPoint**: 单个点云点，表示云坐标系中该点的空间位置及颜色。





- **PointCloudOverrides**: 关联的设置类, 指定由视图存储的、用于 **PointCloudInstance** 图元或图元中扫描的图形重写。

### 1. 创建点云 (Creating a Point Cloud)

要在 Revit 文件中新建点云, 先创建 **PointCloudType**, 再用它创建 **PointCloudInstance**。静态方法 **PointCloudType.Create()** 需要引擎标识符, 正如由第三方将其注册到 Revit 一样, 如果是受支持的文件类型, 则需要该点云文件的文件扩展名。对基于引擎的“非文件”, 它还需要文件名称或标识字符串。代码 5-27 演示了用 rcs 格式文件在 Revit 文件中创建点云, 图 5-8 为文件 32\_cafeteria.rcs 生成的点云。

#### 代码 5-27: 由 rcsrscs 文件创建点云

```
private PointCloudInstance CreatePointCloud (Document doc)
{
    PointCloudType type = PointCloudType.Create (doc, "rcs", "c: \\32_cafeteria.rcs");
    return (PointCloudInstance.Create (doc, type.Id, Transform.Identity));
}
```

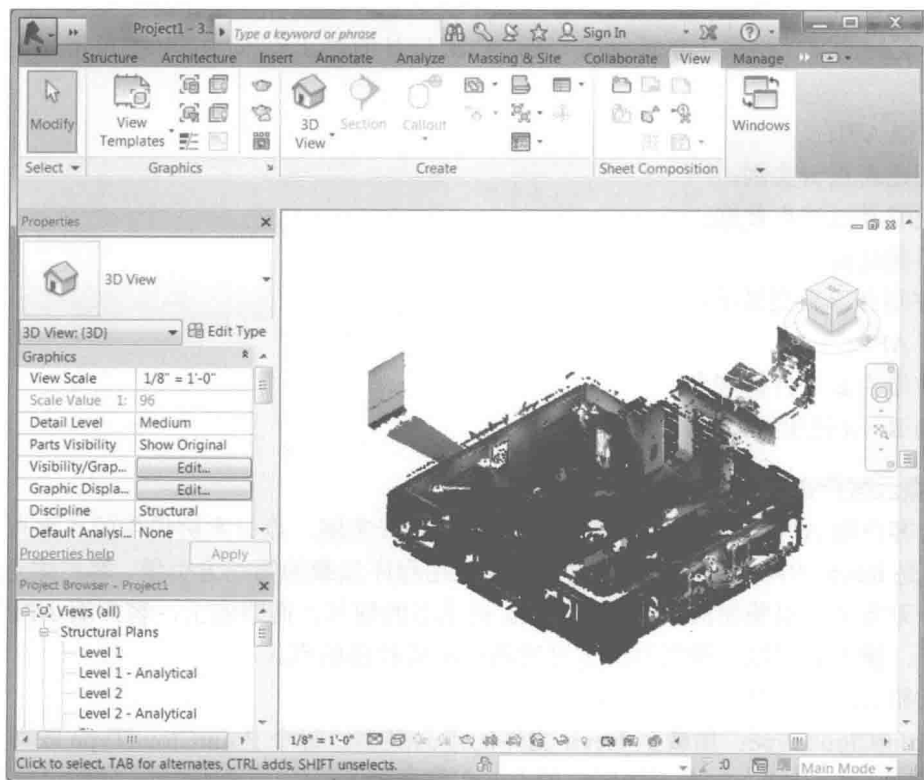


图 5-8 由 32\_cafeteria.rcs 生成的点云

### 2. 访问点云中的点 (Accessing Points in a Point Cloud)

有两种方法来访问点云中的点:

- (1) 直接迭代从 **IEnumerable<CloudPoint>** 接口返回的 **PointCollection** 产生的点。
- (2) 获取该集合中存储点的指针, 在不太稳定的接口中直接访问内存中的点。



无论哪种方式, 从 `PointCloudInstance` 中访问点集合的第一步都是用这种方法: `PointCloudInstance.GetPoints(PointCloudFilter filter, double averageDistance, int numPoints)`。

要注意的是, 作为 Revit 和点云引擎所使用搜索算法的结果, 可能无法返回所需要点的准确数量。

虽然第二种方法直接涉及指针的处理, 有可能在遍历点的大缓冲区时提升性能, 不过仅可从 C# 和 C++/CLI 用此方法。

代码 5-28 和代码 5-29 分别演示了如何使用这两种方法读取点云中的点。

#### 代码 5-28: 通过迭代来读取点云中的点

```
private void GetPointCloudDataByIteration (PointCloudInstance pcInstance, PointCloudFilter pointCloudFilter)
{
    // read points by iteration
    double averageDistance = 0.001;
    PointCollection points = pcInstance.GetPoints(pointCloudFilter, averageDistance, 10000); // Get points. Number of points
    is determined by the needs of the client
    foreach (CloudPoint point in points)
    {
        // Process each point
        System.Drawing.Color color = System.Drawing.ColorTranslator.FromWin32 (point.Color);
        String pointDescription = String.Format ("({0}, {1}, {2}, {3}", point.X, point.Y, point.Z, color.ToString());
    }
}
```

#### 代码 5-29: 通过指针来读取点云中的点

```
public unsafe void GetPointCloudDataByPointer (PointCloudInstance pcInstance, PointCloudFilter pointCloudFilter)
{
    double averageDistance = 0.001;
    PointCollection points = pcInstance.GetPoints (pointCloudFilter, averageDistance, 10000);
    CloudPoint* pointBuffer = (CloudPoint*) points.GetPointBufferPointer().ToPointer();
    int totalCount = points.Count;
    for (int numberOfPoints = 0; numberOfPoints < totalCount; numberOfPoints++)
    {
        CloudPoint point = * (pointBuffer + numberOfPoints);
        // Process each point
        System.Drawing.Color color = System.Drawing.ColorTranslator.FromWin32 (point.Color);
        String pointDescription = String.Format ("({0}, {1}, {2}, {3}", point.X, point.Y, point.Z, color.ToString());
    }
}
```

### 3. 过滤器 (Filters)

过滤器用于限制读取点时的搜索量, 同时也用于控制点云数据的显示。`PointCloudFilter` 可以基于平面边界集合创建。过滤器将检查某个点是否位于如同平面法线正向所示的每个输入平面的“正”面。因此, 这种过滤器隐式定义一个体量, 它对应所有平面的正半空间的交集。这个体量无需闭合, 但它始终是个凸集。



可控制点云数据显示的指定过滤器为: `PointCloudInstance.SetSelectionFilter()`。

滤出点的显示将基于其属性值: `PointCloudInstance.FilterAction`。

如果它被设置为 `None`, 则该选择过滤器将被忽略。如果被设置为 `Highlight`, 通过过滤器的点将高亮显示。如果被设置为 `Isolate`, 则仅通过过滤器的点可见。

代码 5-30 演示了将基于其边界框高亮显示点云中点的子集。

#### 代码 5-30: 通过指针读取点云的点数据

```
// Filter will match 1/8 of the overall point cloud
// Use the bounding box (filter coordinates are in the coordinates of the model)
BoundingBoxXYZ boundingBox = pointCloudInstance.get_BoundingBox (null);
List<Plane> planes = new List<Plane>();
XYZ midpoint = (boundingBox.Min + boundingBox.Max) / 2.0;

// X boundaries
planes.Add (app.Create.NewPlane (XYZ.BasisX, boundingBox.Min));
planes.Add (app.Create.NewPlane (-XYZ.BasisX, midpoint));

// Y boundaries
planes.Add (app.Create.NewPlane (XYZ.BasisY, boundingBox.Min));
planes.Add (app.Create.NewPlane (-XYZ.BasisY, midpoint));

// Z boundaries
planes.Add (app.Create.NewPlane (XYZ.BasisZ, boundingBox.Min));
planes.Add (app.Create.NewPlane (-XYZ.BasisZ, midpoint));

// Create filter
PointCloudFilter filter = PointCloudFilterFactory.CreateMultiPlaneFilter (planes);
pointCloudInstance.FilterAction = SelectionFilterAction.Highlight;
```

图 5-9 为代码 5-30 在一个小的管道点云数据上运行的结果。

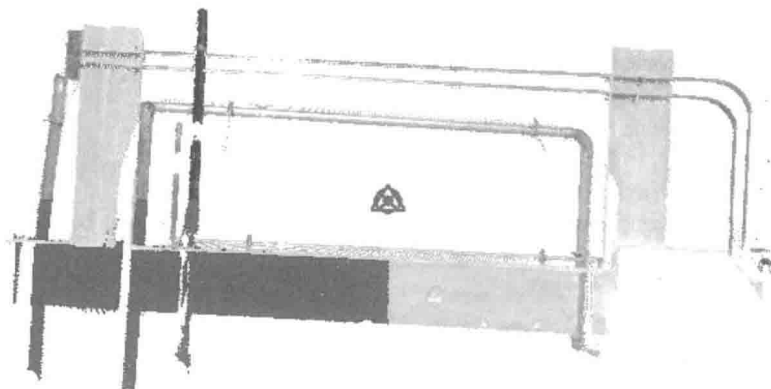


图 5-9 带有选择过滤器的点云

`Selection.PickBox()`方法调用通用的双击编辑器, 让用户在屏幕上指定某个矩形区域, 利用生成的 `PickedBox` 结合使用 `PointCloudFilter`, 来生成过滤器的平面边界。



#### 4. 扫描 (Scans)

.rcp 文件可包含多个扫描。PointCloudInstance.GetScans()方法返回扫描名称列表,可用于独立设置 PointCloudInstance 中每个扫描的可见性和固定的颜色重写。PointCloudInstance.ContainsScan()指示给定的扫描名称是否包含于该点云实例中,而 PointCloudInstance.GetScanOrigin()将返回给定扫描在模型坐标系中的原点。

#### 5. 替换 (Overrides)

分配到给定视图的点云重写设置可使用 Revit API 修改。这些设置对应 Revit UI 中可见性/图形替换任务面板的点云选项卡上的设置。替换可应用于整个点云实例,或该实例中的特定扫描。用于替换的选项包括设置点云实例中扫描的可见性、将其设置为固定的颜色或基于高程、法线或亮度的颜色渐变。属性 PointCloudInstance.SupportsOverrides 识别点云支持哪种替换设置(基于.rcp 或 .rcs 格式文件的云)。

设置点云替换所涉及的类如下:

- PointCloudOverrides: 用于获取或设置点云实例的 PointCloudOverrideSettings。
- PointCloudOverrideSettings: 用于获取或设置点云实例或实例中某个扫描的可见性、着色模式以及 PointCloudColorSettings。
- PointCloudColorSettings: 用于对点云实例图元或其中某个扫描件中某些着色模式指定特定的颜色。若 PointCloudColorMode 为 “NoOverride” 或 “Normals” 则不适用。

### 5.10.2 点云引擎 (Point Cloud Engine)

可以实现自定义点云引擎来向 Revit 提供点云。

点云引擎可以基于文件或不基于文件。基于文件的引擎实现要求每个点云被映射到硬盘上的单个文件。通过选择文件扩展名匹配引擎标识符的点云文件,Revit 允许用户直接在文件中创建新的点云实例。这些文件被视为 Revit 的外部链接,如有必要,可从管理链接对话框重加载、重映射。

非基于文件的引擎实现可以从任何地方(数据库、一个服务器,或一个更大的聚合文件的一部分)得到点云。因为没有用户可以选择的文件,Revit 用户界面将不允许用户创建这种类型的点云;相反,引擎供应商 PointCloudType.Create()和 PointCloudInstance.Create()提供自定义命令来创建和放置此类型的点云。管理链接对话框将显示此类型的点云,但由于没有点云相关联的文件,用户无法管理、重加载或重映射这种类型的点云。

无论是什么实现类型,自定义引擎实现包括:

- 通过 PointCloudEngineRegistry 注册 Revit IPointCloudEngine 的实现。
- 关于单一点云的属性, IPointCloudAccess 实现将响应来自 Revit 的查询。
- 当有请求时, IPointSetIterator 的实现将返回点集给 Revit。
- 为了提供点云的点给 Revit, 有两个 ReadPoints()方法必须实现:
  - IPointCloudAccess.ReadPoints(): 无论是从 Revit 还是 API, 调用一次可提供一个单一点集。Revit 在一些显示活动期间使用它,包括选择预先加亮。API 客户端也可以通过 PointCloudInstance.GetPoints(), 直接调用此方法。



- `IPointSetIterator.ReadPoints()`: 它提供了一个子集, 作为一个较大的迭代云中点的一部分。Revit 在点云正常显示期间使用此方法; 该方法会反复请求大量的点数据, 直到获得足够的点或直到显示的某些部分被更改。引擎实现必须在任何给定点集的迭代期间跟踪已返回到 Revit 的那些点。

注册和实现基于文件的和非基于文件的点云引擎的一个完整示例, 包含在 Revit API SDK 示例目录下 `PointCloudEngine` 文件夹中。

## 5.11 分析 (Analysis)

### 5.11.1 能量数据 (Energy Data)

`EnergyDataSettings` 对象表示 Revit 项目中的 gbXML 参数。要在 Revit UI 中查看参数, 可从管理选项卡上项目设置面板中选择项目信息, 将会出现项目信息对话框 (图 5-10)。

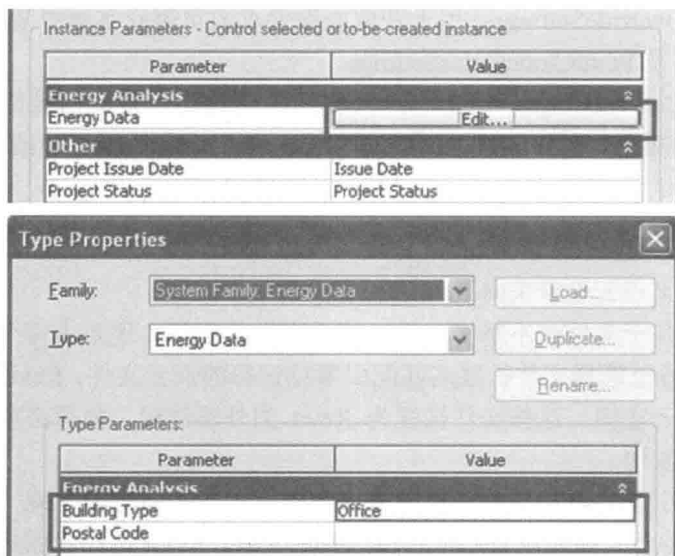


图 5-10 能量数据设置

`EnergyDataSettings` 对象是由图元基对象派生。在每个项目中它是唯一的, 类似于项目信息。虽然 `EnergyDataSettings` 是图元类的子类, 但除了名称、ID、唯一 ID 和参数, 大部分从图元继承的成员返回 “null” 或 “空集”。

代码 5-31 为 `EnergyDataSettings` 类的使用示例。在调用命令后, 结果会显示在任务对话框中。

#### 代码 5-31: 使用 `EnergyDataSettings` 类

```
public void GetInfo_EnergyData (Document document)
{
    EnergyDataSettings energyData = EnergyDataSettings.GetFromDocument (document);
```



```

if (null != energyData)
{
    string message = "energyData : ";
    message += "\nBuildingType : " + energyData.BuildingType;
    TaskDialog.Show ("Revit", message);
}
}

```

### 5.11.2 分析可视化 (Analysis Visualization)

Revit API 提供一种机制, 以方便外部分析应用程序将计算的成果显示在 Revit 模型中。SpatialFieldManager 类是将分析成果传回给 Revit 的主要的类。它用于创建、删除和修改存储分析成果的“容器”。AnalysisResultSchema 类包含分析成果的所有信息, 如用于成果可视化的说明、名称及所有单位乘数。SpatialFieldManager 可注册多个 AnalysisResultSchemas。

AnalysisDisplayStyle 类可用来控制成果的外观。插件的 AnalysisDisplayStyle 创建和修改是可选的; 对分析成果的展示, 使用者可以获得与 Revit UI 同样的掌控。

Revit API 支持的数据模型需要在某指定点集上详细说明分析成果, 并且计算出每个点上一个或多个不同的数量 (“测量值”)。在所有模型点的测量数量必须相同。成果数据是暂态的; 它只存储在模型中, 直到文件关闭。如果模型保存、关闭以及重新打开, 则分析成果将不复存在。

#### 1. 分析成果管理器 (Manager for Analysis Results)

使用静态方法 SpatialFieldManager.CreateSpatialFieldManager(), SpatialFieldManager 可以新加到视图中。视图相关的管理器只能有一个。若视图已有一个 SpatialFieldManager, 它可以用静态方法 GetSpatialFieldManager() 来检索。

CreateSpatialFieldManager() 需要一个为各点求出测量数量的参数。该数字定义了与每个求出成果的点相关的成果值数量。例如, 若计算了一年中每个月的平均太阳能, 则每个点都有 12 个对应的值。

要将分析成果添加到视图中, 请调用 AddSpatialFieldPrimitive() 创建一个新的分析成果容器。此方法有 4 个重载来创建相关的容器:

- 一个参照 (对曲线或面)。
- 一个曲线和一个转换。
- 一个面和一个转换。
- 非 Revit 几何: 当创建非 Revit 几何相关的许多数据点时, 为提高性能, 建议创建每个容器不超过 500 个点的多个容器, 而不是一个包含所有点的大型容器。

一个典型的转换重载应用是定位成果数据对 Revit 模型中几何体的偏移量, 例如, 在楼面以上 3 英尺。

AddSpatialFieldPrimitive() 方法在 SpatialFieldManager 内返回容器的唯一整数标识符, 以便随后用于标识该容器来将其删除 [ RemoveSpatialFieldPrimitive() ] 或修改 [ UpdateSpatialFieldPrimitive() ]。

注意: AddSpatialFieldPrimitive() 方法所创建的是一个空的分析成果容器。必须调用 UpdateSpatialFieldPrimitive() 按顺序填充带有点和值的分析成果数据, 如下节 “创建分析结



果数据”中所示。

UpdateSpatialFieldPrimitive() 方法需要一个已随 SpatialFieldManager 注册的 AnalysisResultSchema 唯一索引。AnalysisResultSchema 包含有关分析成果的信息, 如用于成果可视化的名称、说明、所有单位名称和乘数。下面的代码 5-32 演示了如何创建一个新的 AnalysisResultSchema 并设置其单位。

#### 代码 5-32: AnalysisResultsSchema

```
ICollection<string> unitNames = new List<string>( );
unitNames.Add ( "Feet" );
unitNames.Add ( "Inches" );
ICollection<double> multipliers = new List<double>( );
multipliers.Add ( 1 );
multipliers.Add ( 12 );

AnalysisResultSchema resultSchema = new AnalysisResultSchema ( "Schema Name", "Description" );

resultSchema.SetUnits ( unitNames, multipliers );
```

一旦 AnalysisResultschema 已配置, 即需要使用 SpatialFieldManager.RegisterResult() 方法来注册。它将返回成果的唯一索引。在注册后, 以此唯一索引使用 GetResultSchema() 和 SetResultSchema() 可获取和更改其成果。

#### 2. 创建分析成果数据 (Creating Analysis Results Data)

一旦向 SpatialFieldManager 添加了一个容器, 即可使用 UpdateSpatialFieldPrimitive() 方法, 创建分析成果并将其添加到分析成果容器。该方法需要已求出结果的一组域点 (FieldDomainPoints) 和每个点的一组值 (FieldValues)。FieldValues 的数量必须对应于域点的数量。然而, 每个域点可以有一组值, 每个值对应这一点上各自的“测量值”。

代码 5-33 演示了如何在用户所选图元面上创建一组简单的分析成果。SDK “SpatialFieldGradient” 演示了一个更复杂的用例, 其中每个点都有多个关联值。

#### 代码 5-33: 创建分析成果

```
Document doc = commandData.Application.ActiveUIDocument.Document;
UIDocument uiDoc = commandData.Application.ActiveUIDocument;

SpatialFieldManager sfm = SpatialFieldManager.GetSpatialFieldManager ( doc.ActiveView );
if ( null == sfm )
{
    sfm = SpatialFieldManager.CreateSpatialFieldManager ( doc.ActiveView, 1 );
}

Reference reference = uiDoc.Selection.PickObject ( ObjectType.Face, "Select a face" );
int idx = sfm.AddSpatialFieldPrimitive ( reference );

Face face = doc.GetElement ( reference ).GetGeometryObjectFromReference ( reference ) as Face;

ICollection<UV> uvPts = new List<UV>( );
BoundingBoxUV bb = face.GetBoundingBox( );
```



```

UV min = bb.Min;
UV max = bb.Max;
uvPts.Add (new UV (min.U, min.V));
uvPts.Add (new UV (max.U, max.V));

FieldDomainPointsByUV pnts = new FieldDomainPointsByUV (uvPts);

List<double> doubleList = new List<double>( );
IList<ValueAtPoint> valList = new List<ValueAtPoint>( );
doubleList.Add (0);
valList.Add (new ValueAtPoint (doubleList));
doubleList.Clear( );
doubleList.Add (10);
valList.Add (new ValueAtPoint (doubleList));

FieldValues vals = new FieldValues (valList);

AnalysisResultSchema resultSchema = new AnalysisResultSchema ("Schema Name", "Description");
int schemaIndex = sfm.RegisterResult (resultSchema);
sfm.UpdateSpatialFieldPrimitive (idx, pnts, vals, schemaIndex);

```

### 3. 分析成果显示 (Analysis Results Display)

AnalysisDisplayStyle 类可用于控制如何在视图中显示分析成果。静态方法 CreateAnalysisDisplayStyle() 可以创建表面着色、文本标记、变形图、图表或矢量等多种显示样式(图 5-11)。无论何种样式, 其颜色和图例都可设置。

创建了一个新的 AnalysisDisplayStyle, 即可使用 View.AnalysisDisplayStyleId 来指派视图的样式。虽然分析成果未与文件一起保存, 但模型中保存有视图的分析显示样式及其分配。

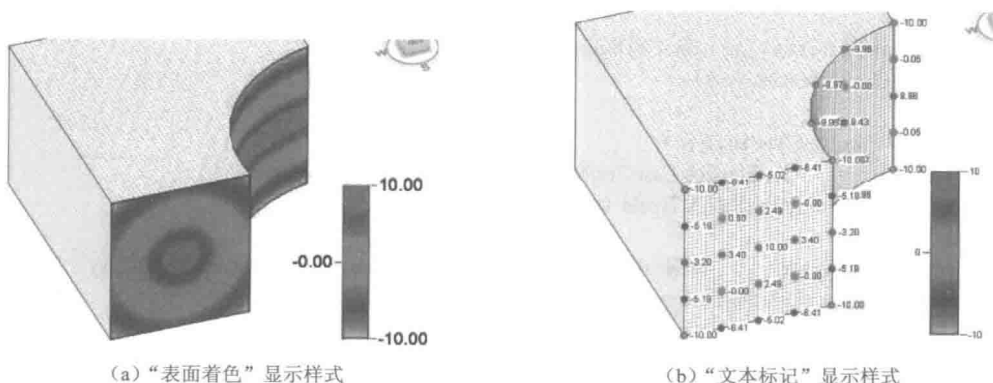


图 5-11 视图显示样式

代码 5-34 演示了如何创建一个新的表面着色分析显示样式(如果尚未在文件中找到), 然后将它指派给当前视图。



**代码 5-34: 设置视图的分析显示样式**

```

Document doc = commandData.Application.ActiveUIDocument.Document;

AnalysisDisplayStyle analysisDisplayStyle = null;
// Look for an existing analysis display style with a specific name
FilteredElementCollector collector1 = new FilteredElementCollector ( doc );
ICollection<Element> collection =
    collector1.OfClass ( typeof ( AnalysisDisplayStyle ) ) .ToElements();
var displayStyle = from element in collection
                    where element.Name == "Display Style 1"
                    select element;

// If display style does not already exist in the document, create it
if ( displayStyle.Count() == 0 )
{
    AnalysisDisplayColoredSurfaceSettings coloredSurfaceSettings = new AnalysisDisplayColoredSurfaceSettings ( );
    coloredSurfaceSettings.ShowGridLines = true;

    AnalysisDisplayColorSettings colorSettings = new AnalysisDisplayColorSettings ( );
    Color orange = new Color ( 255, 205, 0 );
    Color purple = new Color ( 200, 0, 200 );
    colorSettings.MaxColor = orange;
    colorSettings.MinColor = purple;
    AnalysisDisplayLegendSettings legendSettings = new AnalysisDisplayLegendSettings ( );
    legendSettings.NumberOfSteps = 10;
    legendSettings.Rounding = 0.05;
    legendSettings.ShowDataDescription = false;
    legendSettings.ShowLegend = true;

    FilteredElementCollector collector2 = new FilteredElementCollector ( doc );
    ICollection<Element> elementCollection = collector2.OfClass ( typeof ( TextNoteType ) ) .ToElements ( );
    var textElements = from element in collector2
                       where element.Name == "LegendText"
                       select element;

    // if LegendText exists, use it for this Display Style
    if ( textElements.Count() > 0 )
    {
        TextNoteType textType =
            textElements.Cast<TextNoteType>( ).ElementAt<TextNoteType> ( 0 );
        legendSettings.SetTextTypeId ( textType.Id, doc );
    }
    analysisDisplayStyle = AnalysisDisplayStyle.CreateAnalysisDisplayStyle ( doc , "Display Style 1" ,
    coloredSurfaceSettings, colorSettings, legendSettings );
}
else
{
    analysisDisplayStyle =
        displayStyle.Cast<AnalysisDisplayStyle>( ).ElementAt<AnalysisDisplayStyle> ( 0 );
}
// now assign the display style to the view
doc.ActiveView.AnalysisDisplayStyleId = analysisDisplayStyle.Id;

```



#### 4. 更新分析成果 (Updating Analysis Results)

Revit 分析框架不会自动更新结果, 对 Revit 模型的任何更改都可能使结果无效。

当 Revit 模型已发生更改、之前的计算结果可能无效, 需要重新计算时, 为了使结果能够更新, API 开发人员应使用动态模型更新触发器或订阅 `DocumentChanged` 事件等待更新通知。一个连同分析可视化的动态模型更新演示的例子, 请参阅 Revit sdk 中的 “DistanceToSurfaces” 示例代码。

### 5.11.3 概念能量分析 (Conceptual Energy Analysis)

Revit API 提供对 Revit 创建的图元和对对象的访问, 来执行概念设计模型的能量分析。下面的类可操作这些数据:

- `MassEnergyAnalyticalModel`: 与能量分析模型数据及几何体有关的体量实例。
- `MassLevelData`: 概念建筑模型中一个可占用楼层 (体量楼层) 的概念表示。
- `MassSurfaceData`: 含有有关 `MassEnergyAnalyticalModel` 图元中某个面的属性及其他数据。
- `MassZone`: 建筑的单独加热及冷却子体量的概念表示。
- `ConceptualConstructionType`: 以 Revit BIM 模型和绿色建筑工作室/绿色建筑可扩展标记语言 (gBXML) 都能接受的方式, 描述概念上的物理、结构和能量属性。
- `ConceptualSurfaceType`: 表示指派给体量几何面的概念性的 BIM 对象类别。

除了这些类以外, 还有一个 `Document.Export()` 重载, 需要一个 `MassGBXMLExportOptions` 参数, 它输出一个只包含概念能量分析图元体量图元的 gBXML 文件。

#### 1. 体量能量分析模型 (MassEnergyAnalyticalModel)

相关概念能量分析的主类是 `MassEnergyAnalyticalModel`。该类把体量实例与能量分析模型数据及几何相关联。几何体一开始作为其关联的体量实例几何的一个副本, 并根据能量分析模型的要求作修改。`MassEnergyAnalyticalModel.GetMassEnergyAnalyticalModelIdForMassInstance()` 静态方法对于给定的体量实例将返回 `MassEnergyAnalyticalModel` 的图元 ID。

#### 2. 体量层数据 (MassLevelData)

体量层数据是在 Revit 项目中通过将某个特定层与某个特定体量图元相关联来定义。这可以通过使用 `MassInstanceUtils.AddMassLevelDataToMassInstance()` 方法来完成。`MassLevelData` 报告度量指标, 如楼层面积, 相关的概念空间规划。`MassLevelData` 包含信息如 `ConceptualConstructionType`, 作为概念能量分析模型的一部分。体量层数据几何是通过合并所有体量几何为单个几何, 然后取得与 `MassLevelData` 的层相交的区域来确定。

#### 3. 体量分区 (MassZones)

由同一体量族实例相关联的体量层数据与体量能量分析模型的几何形状相交, 来将体量能量分析模型划分成片状, 可以创建体量分区。使用 `GetMassZoneIds()` 方法, 可以检索与体量能量分析模型相关联的体量分区的图元 ID。

#### 4. 体量表面数据 (MassSurfaceData)

从体量能量分析模型, 可以获取对它有意义的所有表面的参照。利用这些参照, 使用



`GetMassSurfaceDataIdForReference()`方法,可以得到与每一个面相关的体量表面数据。体量表面数据含有与体量能量分析模型图元中某个面有关的属性和其他数据,如材料值、自动生成的图元尺寸(如窗台高度、天窗宽度)。

体量表面数据还含有与参照面相关的 `ConceptualConstructionType` 的 ID。`ConceptualConstructionType` 以 Revit BIM 模型和绿色建筑工作室/绿色建筑可扩展标记语言(gBXML)都能接受的方式,描述概念上的物理、结构和能量属性。该类含有大量的静态方法,用来获取给定文件中建筑物各个方面(如墙和窗)的 `ConceptualConstructionType` 的图元 ID。

体量表面数据的另一个属性是 `CategoryIdForConceptualSurfaceType`,它提供体量子类别的图元 ID,用于其 `ConceptualSurfaceType`。`ConceptualSurfaceType` 表示指派给体量几何面的概念性的 BIM 对象类别。每个体量表面子类别对应一个 `ConceptualSurfaceType` 图元。使用静态方法 `ConceptualSurfaceType.GetByMassSubCategoryId()`,可以从体量类别 ID 获得每个体量表面数据的 `ConceptualSurfaceType`。

当在 Revit 项目中启用概念能量分析时,体量面将被分配到这些 `ConceptualSurfaceType` 与之关联的体量类别的子类别。默认 `ConceptualConstructionType` 与 `ConceptualSurfaceType` 相关联。此默认 `ConceptualConstructionType` 和相应的子类别一起被分配给体量面。更改与 `ConceptualSurfaceType` 关联的默认 `ConceptualConstructionType` 将更新该子类别的所有的体量面的 `ConceptualConstruction` 类型,因为用户并未专门为其提供替代值。

#### 5. 体量实例实用工具 (MassInstanceUtils)

`MassInstancesUtils` 工具类提供静态方法来获取有关体量实例的信息,如通过体量实例表示的可占用楼面总面积或建筑总量,以及如前面所讨论,创建体量层数据(体量楼层)来使某个层与体量实例相关联。

#### 5.11.4 能量分析详细模型 (Detailed Energy Analysis Model)

Autodesk. Revit. DB. Analysis 命名空间包含一些类,可获取和分析项目的详细能量分析模型的内容。输出 gbXML 文件及加热和冷却负载特性,可从建筑物的物理模型产生热分析模型。热分析模型由代表实际建筑体积图元的空间、区域和平面组成。

有关详细的能量分析模型的类有:

- `EnergyAnalysisDetailModel`。
- `EnergyAnalysisDetailModelOptions`。
- `EnergyAnalysisOpening`。
- `EnergyAnalysisSpace`。
- `EnergyAnalysisSurface`。
- `Polyloop`。

使用静态方法 `EnergyAnalysisDetailModel.Create()`来创建和填充能量分析模型。使用 `EnergyAnalysisDetailModelOptions` 来设置适当的选项。所生成的模型总是返回到全局坐标系中,但 `TransformModel()`方法可根据地面、共享坐标和真北方向转换模型中的所有面。

创建能量分析详细模型的可用选项包括:

- 能量分析模型的计算等级:非计算的, `FirstLevelBoundaries`,意指分析空间和区域,



SecondLevelBoundaries, 意指分析表面, 或 Final, 意指构造、明细表及非图形数据。

- 多格窗是否输出为着色表面。
- 是否包括着色表面。
- 是否简化幕墙系统: 当为 “true”, 则不管系统中有多少嵌板, 都将幕墙/系统输出为一个大的单一窗/开口。

代码 5-35 演示了由物理模型创建一个新的能量分析的详细模型, 然后显示模型中每个空间每个表面的原始图元。

#### 代码 5-35: 能量分析详细模型

```
// Collect space and surface data from the building's analytical thermal model
EnergyAnalysisDetailModelOptions options = new EnergyAnalysisDetailModelOptions();
options.Tier = EnergyAnalysisDetailModelTier.Final; // include constructions, schedules, and non-graphical data in the
computation of the energy analysis model
EnergyAnalysisDetailModel eadm = EnergyAnalysisDetailModel.Create(doc, options); // Create a new energy analysis detailed
model from the physical model
IList<EnergyAnalysisSpace> spaces = eadm.GetAnalyticalSpaces();
StringBuilder builder = new StringBuilder();
builder.AppendLine("Spaces: " + spaces.Count);
foreach (EnergyAnalysisSpace space in spaces)
{
    builder.AppendLine(">> " + space.Name + " related to " + space.SpatialElementId);
    IList<EnergyAnalysisSurface> surfaces = space.GetAnalyticalSurfaces();
    builder.AppendLine("has " + surfaces.Count + " surfaces.");
    foreach (EnergyAnalysisSurface surface in surfaces)
    {
        builder.AppendLine("+++ Surface from " + surface.OriginatingElementDescription);
    }
}

TaskDialog.Show("EAM", builder.ToString());
```

创建能量分析详细模型之后, 与其相关的空间、开口和表面可随 GetAnalyticalOpenings()、GetAnalyticalSpaces()、GetAnalyticalShadingSurfaces()和 GetAnalyticalSurfaces()方法来检索。

在完成分析成果之后, 请务必调用 EnergyAnalysisDetailModel.Destroy() 以清理 Revit 数据库。

##### 1. 能量分析空间 (EnergyAnalysisSpace)

从能量分析空间可以检索由墙中心平面和屋顶及楼板顶部平面所定义封闭体积边界的能量分析表面的集合。另外, GetClosedShell()也可检索一个 Polyloops 集合, 这是个平面多边形, 定义一个由内部边界面测量出的封闭体积。对于二维情况, 使用 GetBoundary()返回一个 Polyloops 集合, 代表空间的二维边界, 定义一个由内部边界面测量出的封闭面积。

EnergyAnalysisSpace 类还具有一些属性, 用于访问有关分析空间的信息, 如 AnalyticalVolume、名称和面积。



## 2. 能量分析表面 (EnergyAnalysisSurface)

从能量分析空间可以检索与表面关联的原始分析空间以及二次邻近分析空间。`GetAnalyticalOpenings()`方法将检索一个在该面上的所有分析开口的集合。`GetPolyloop()`方法可获取描述表面几何形状的平面多边形, 如同 `gbXML` 中所描述的一样。

`EnergyAnalysisSpace` 类有许多属性, 以提供更多关于分析表面的信息, 如高度、宽度、角点 (从外部查看分析矩形几何的左下角坐标), 以及一个原始图元描述。

表面类型既可用作 `EnergyAnalysisSurfaceType` 也可用作 `gbXMLSurfaceType`。`gbXML` 表面类型属性是由源图元和邻近空间的数量来决定。可能的类型见表 5-6。

表 5-6 能量分析表面类型

类型	源图元和邻近空间
Shade	无关联的源图元和邻近空间
Air	无关联的源图元, 至少有一个邻近空间
ExteriorWall	源图元为墙或幕墙, 有一个邻近空间
InteriorWall	源图元为墙或幕墙且: 有两个邻近空间, 或者类型函数参数设置为 “Interior” 或 “CoreShaft”
UndergroundWall	源图元为墙或幕墙, 如果在地面以下, 则有一个邻近空间
SlabOnGrade	源图元为楼板, 有一个邻近空间
RaisedFloor	源图元为楼板, 有一个邻近空间, 且在地面以上
UndergroundSlab	源图元为楼板, 有一个邻近空间, 且在地面以下
InteriorFloor	源图元为楼板, 且有两个邻近空间, 或者类型函数参数设置为 “Interior”
Roof	源图元为屋顶或天花板, 有一个邻近空间
UndergroundCeiling	源图元为屋顶或天花板, 有一个邻近空间, 且在地面以下
Ceiling	源图元为屋顶或天花板, 有两个邻近空间

## 3. 能量分析开口 (EnergyAnalysisOpening)

从能量分析开口可以检索关联的父分析表面图元。`GetPolyloop()`方法返回为平面多边形的开口几何形状。

很多属性可用于获取有关分析开口的信息, 如高度、宽度、角点和名称。与分析表面相似, 作为一个简单的 `EnergyAnalysisOpeningType` 枚举或作为 `gbXMLOpeningType` 属性, 可以获得分析开口的类型。开口类型的确定是基于开口的族类别以及它包含在什么图元中, 见表 5-7。

表 5-7 能量分析开口类型

Type	族类别或包含的图元	Type	族类别或包含的图元
OperableWindow	窗	FixedWindow	幕墙嵌板上的开口
NonSlidingDoor	门	Air	Openings 类别的开口
FixedSkylight	屋顶上的开口		



## 5.12 地点和位置 (Place and Locations)

每个建筑都有一个世界上唯一的地点，因为纬度和经度是唯一的。此外，某个建筑可以有多个与其他建筑物的相对位置。Revit 平台 API 场地命名空间使用某些类来保存 Revit 项目的地理位置信息。

注意：Revit 平台 API 未公开场地菜单功能。仅场地命名空间提供相应的位置选项功能，在管理选项卡的项目位置面板上可以找到（图 5-12）。

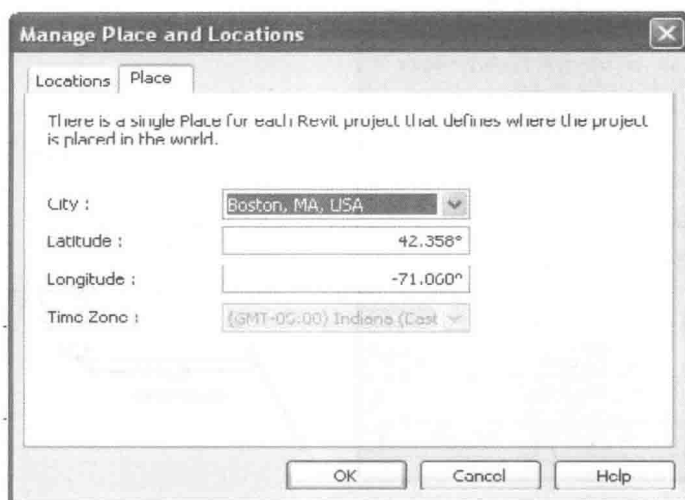


图 5-12 项目地点

### 1. 地点 (Place)

在 Revit 平台 API 中，`SiteLocation` 类包含地点信息，包括纬度、经度和时区。此信息标识项目位于世界上哪个地方。当设置经度或纬度时，请注意：

(1) Revit 会尝试将坐标匹配到它已知的某个城市，如果找到匹配项，则将据此设置名称。

(2) 使用 `SunAndShadowSettings.CalculateTimeZone()`，Revit 会尝试自动调整时区值以匹配新的经度和纬度设置值。某些边界情况下，所算出的时区可能不正确，如有必要，可以直接设置 `TimeZone` 属性。

### 2. 城市 (City)

城市是一个对象，该对象包含世界上已知城市的地理位置信息。它包含经度、纬度和时区信息。城市列表可由应用程序对象中的 `Cities` 属性来检索。新的城市不能添加到 Revit 现有列表中。Revit 平台 API 也不会公开当前项目所在的城市。

### 3. 项目位置 (ProjectLocation)

一个项目只有一个地点，它是地球上的绝对位置。然而，它可以有不同的与周围项目的相对位置。根据使用的坐标和原点，一个项目中可以有多个项目位置对象。



默认情况下每个 Revit 项目至少包含一个内部的命名位置。它是活动项目的位置，可以使用 `Document.ActiveProjectLocation` 属性来检索。使用 `Document.ProjectLocations` 属性可以检索所有现有的项目位置对象。

#### 4. 项目定位 (Project Position)

项目定位也是一个对象，该对象代表一个地理偏移和旋转。它通常是由项目位置对象来获取和设置地理信息。图 5-13 显示了在改变项目位置地理旋转后的结果和同一点的坐标。然而，您无法直接看到改变项目位置地理偏移的结果。图 5-14 为地理偏移和旋转示意图。



图 5-13 点坐标

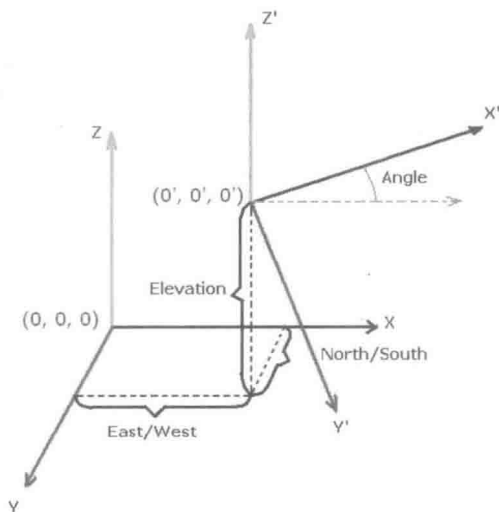


图 5-14 地理偏移和旋转示意图

注意: East 表示位置逆时针旋转; West 表明位置顺时针旋转。如果角度值介于  $180^\circ \sim 360^\circ$ , Revit 会自动转换。例如, 如果选择 East、角度键入  $200^\circ$ , Revit 会把它转换为 West  $160^\circ$ 。

代码 5-36 演示了如何检索项目位置对象。

#### 代码 5-36: 检索项目位置对象

```
public void ShowActiveProjectLocationUsage (Autodesk.Revit.DB.Document document)
{
    // Get the project location handle
    ProjectLocation projectLocation = document.ActiveProjectLocation;

    // Show the information of current project location
    XYZ origin = new XYZ (0, 0, 0);
    ProjectPosition position = projectLocation.get_ProjectPosition (origin);
    if (null == position)
    {
        throw new Exception ("No project position in origin point.");
    }

    // Format the prompt string to show the message.
    String prompt = "Current project location information: \n";
}
```



```

prompt += "\n\t" + "Origin point position: ";
prompt += "\n\t\t" + "Angle: " + position.Angle;
prompt += "\n\t\t" + "East to West offset: " + position.EastWest;
prompt += "\n\t\t" + "Elevation: " + position.Elevation;
prompt += "\n\t\t" + "North to South offset: " + position.NorthSouth;

// Angles are in radians when coming from Revit API, so we
// convert to degrees for display
const double angleRatio = Math.PI / 180;           // angle conversion factor

SiteLocation site = projectLocation.SiteLocation;
prompt += "\n\t" + "Site location: ";
prompt += "\n\t\t" + "Latitude: " + site.Latitude / angleRatio + "°";
prompt += "\n\t\t" + "Longitude: " + site.Longitude / angleRatio + "°";
prompt += "\n\t\t" + "TimeZone: " + site.TimeZone;

// Give the user some information
TaskDialog.Show ("Revit", prompt);
}

```

注：每次只能有一个活动的项目位置。若要看改变项目位置地理偏移和旋转后的结果，请在平面视图属性对话框（图 5-15）中将方向属性从“项目北”更改为“真北”，图 5-16 为调整后的效果图，图 5-17 为相应的项目位置信息。

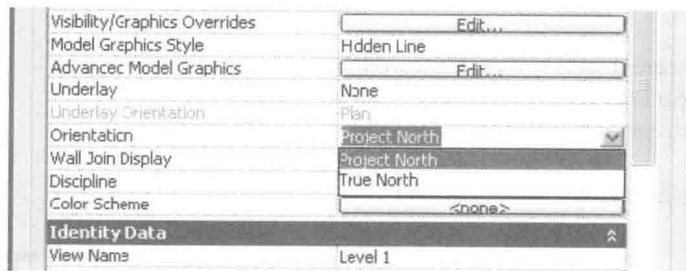


图 5-15 在平面视图属性对话框中设置方向值

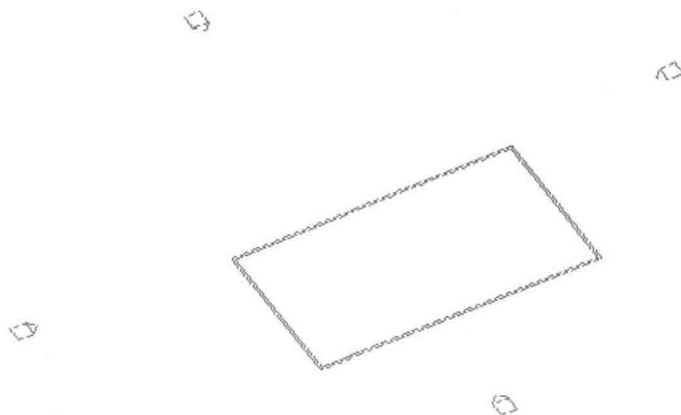


图 5-16 项目从“项目北”旋转 30° 到“真北”



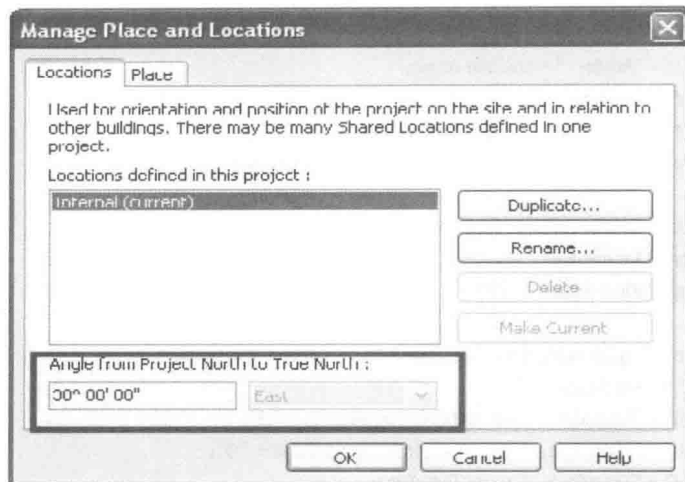


图 5-17 项目位置信息

创建和删除项目位置 (Creating and Deleting Project Locations): 使用 `Duplicate()` 方法, 可通过复制现有项目位置创建新的项目位置。代码 5-37 演示了如何使用 `Duplicate()` 方法创建一个新项目位置。

#### 代码 5-37: 创建项目位置

```
public ProjectLocation DuplicateLocation (Autodesk.Revit.DB.Document document, string newName)
{
    ProjectLocation currentLocation = document.ActiveProjectLocation;
    ProjectLocationSet locations = document.ProjectLocations;
    foreach (ProjectLocation projectLocation in locations)
    {
        if (projectLocation.Name == newName)
        {
            throw new Exception ("The name is same as a project location's name, please change one.");
        }
    }
    return currentLocation.Duplicate (newName);
}
```

代码 5-38 说明了如何从当前项目中删除现有项目位置。

#### 代码 5-38: 删除项目位置

```
public void DeleteLocation (Autodesk.Revit.DB.Document document)
{
    ProjectLocation currentLocation = document.ActiveProjectLocation;
    //There must be at least one project location in the project.
    ProjectLocationSet locations = document.ProjectLocations;
    if (1 == locations.Size)
    {
        return;
    }
}
```



```

string name = "location";
if (name != currentLocation.Name)
{
    foreach (ProjectLocation projectLocation in locations)
    {
        if (projectLocation.Name == name)
        {
            ICollection<Autodesk.Revit.DB.ElementId> elemSet = document.Delete (projectLocation.Id);
            if (elemSet.Count > 0)
            {
                TaskDialog.Show ("Revit", "Project Location Deleted!");
            }
        }
    }
}
}

```

请注意，下列规则适用于删除项目位置：

- 无法删除活动的项目位置，因为项目中必须至少有一个项目位置。
- 如果 ProjectLocationSet 类实例是只读的，则不能删除该项目位置。

## 5.13 工作共享 (Worksharing)

工作共享是一种设计方法，允许多个团队成员同时处理同一个项目模型。当启用工作共享时，Revit 文件可细分为工作集，即项目中的图元的集合。

### 5.13.1 工作集中的图元 (Elements in Worksets)

文件中的每个图元必须且仅属于一个工作集。每个图元都有一个 WorksetId 来标识其所属的唯一工作集。此外，使用 ElementWorksetFilter，有可能获得属于某一指定工作集的文件中所有的图元，见代码 5-39。

#### 代码 5-39: ElementWorksetFilter

```

public void WorksetElements (Document doc, Workset workset)
{
    // filter all elements that belong to the given workset
    FilteredElementCollector elementCollector = new FilteredElementCollector (doc);
    ElementWorksetFilter elementWorksetFilter = new ElementWorksetFilter (workset.Id, false);
    ICollection<Element> worksetElemsfound = elementCollector.WherePasses (elementWorksetFilter).ToElements();

    // how many elements were found?
    int elementsCount = worksetElemsfound.Count;
    String message = "Element count : " + elementsCount;

    // Get name and/or Id of the elements that pass the given filter and show a few of them
    int count = 5;    // show info for 5 elements only
    foreach (Element ele in worksetElemsfound)

```



```
{
    if (null != ele)
    {
        message += "\nElementId : " + ele.Id;
        message += ", Element Name : " + ele.Name;
        if (0 == --count)
            break;
    }
}

Autodesk.Revit.UI.TaskDialog.Show ("ElementsOfWorkset", message);
}
```

工作共享信息，如当前所有者和图元的检出状态，可以使用 `WorksharingUtils` 类来获取（代码 5-40）。它是一个静态类，包含工作共享相关的实用工具函数。

#### 代码 5-40: `WorksharingUtils`

```
public void GetElementWorksharingInfo (Document doc, ElementId elemId)
{
    String message = String.Empty;
    message += "Element Id: " + elemId;

    Element elem = doc.GetElement (elemId);
    if (null == elem)
    {
        message += "Element does not exist";
        return;
    }

    // The workset the element belongs to
    WorksetId worksetId = elem.WorksetId;
    message += ("\nWorkset Id : " + worksetId.ToString());

    // Model Updates Status of the element
    ModelUpdatesStatus updateStatus = WorksharingUtils.GetModelUpdatesStatus (doc, elemId);
    message += ("\nUpdate status : " + updateStatus.ToString());

    // Checkout Status of the element
    CheckoutStatus checkoutStatus = WorksharingUtils.GetCheckoutStatus (doc, elemId);
    message += ("\nCheckout status : " + checkoutStatus.ToString());

    // Getting WorksharingTooltipInfo of a given element Id
    WorksharingTooltipInfo tooltipInfo = WorksharingUtils.GetWorksharingTooltipInfo (doc, elemId);
    message += ("\nCreator : " + tooltipInfo.Creator);
    message += ("\nCurrent Owner : " + tooltipInfo.Owner);
    message += ("\nLast Changed by : " + tooltipInfo.LastChangedBy);

    Autodesk.Revit.UI.TaskDialog.Show ("GetElementWorksharingInfo", message);
}
```



### 5.13.2 图元所有权 (Element Ownership)

WorksharingUtils 类可用于修改图元和工作集的所有权。CheckoutElements()方法为当前用户获取尽可能多的指定图元的所有权,而 CheckoutWorksets()方法获取尽可能多的指定工作集的所有权。RelinquishOwnership()方法根据指定的 RelinquishOptions,让出当前用户所拥有的图元和工作集。

为获得最佳性能,请在一个大的调用环境中,而不是许多小的调用环境中检索所有图元、工作集和放弃条款的信息。

请注意,当检出某个图元时,Revit 可能检出必要的其他图元,以使请求的图元可编辑。例如,如果图元是在某个组中,则 Revit 将检出整个组。

代码 5-41 可为当前用户检出文件中所有的房间。

#### 代码 5-41: 检出图元

```
void CheckoutAllRooms (Document doc)
{
    FilteredElementCollector collector = new FilteredElementCollector (doc);
    ICollection<ElementId> rooms = collector.WherePasses (new RoomFilter()).ToElementIds();
    ICollection<ElementId> checkoutelements = WorksharingUtils.CheckoutElements (doc, rooms);
    TaskDialog.Show ("Checked out elements", "Number of elements checked out: " + checkoutelements.Count);
}
```

代码 5-42 演示了如何检出所有视图工作集。

#### 代码 5-42: 检出工作集

```
void CheckoutAllViewWorksets (Document doc)
{
    FilteredWorksetCollector collector = new FilteredWorksetCollector (doc);

    // find all view worksets
    collector.OfKind (WorksetKind.ViewWorkset);
    ICollection<WorksetId> viewworksets = collector.ToWorksetIds();
    ICollection<WorksetId> checkoutworksets = WorksharingUtils.CheckoutWorksets (doc, viewworksets);
    TaskDialog.Show ("Checked out worksets", "Number of worksets checked out: " + checkoutworksets.Count);
}
```

### 5.13.3 打开工作共享文件 (Opening a Workshared Document)

Application.OpenDocumentFile (ModelPath, OpenOptions) 方法可用于设置打开工作共享文件相关的选项。除了从中心文件分离或允许本地文件由其所有者以外的用户以只读方式打开选项外,还可以设置相关的工作集选项。当打开一个工作共享文件时,所有系统工作集自动打开,但是用户创建的工作集可以指定为打开或关闭。已打开的工作集中的图元可以扩充和显示。然而,在一个关闭的工作集中,图元是不显示的,以避免扩大。仅在当前会话中打开必要的工作集,可以减少 Revit 的内存占用,对提高性能会有所帮助。

代码 5-43 演示了如何打开一个带有两个指定要打开的工作集的文件。请注意,WorksharingUtils.GetUserWorksetInfo()方法可用于从一个关闭的 Revit 文件中访问工作集信息。

**代码 5-43: 打开工作集文件**

```
void OpenDocumentWithWorksets (Application app, ModelPath projectPath)
{
    try
    {
        // Get info on all the user worksets in the project prior to opening
        IList<WorksetPreview> worksets = WorksharingUtils.GetUserWorksetInfo (projectPath);
        IList<WorksetId> worksetIds = new List<WorksetId>();
        // Find two predetermined worksets
        foreach (WorksetPreview worksetPrev in worksets)
        {
            if (worksetPrev.Name.CompareTo ("Workset1") == 0 ||
                worksetPrev.Name.CompareTo ("Workset2") == 0)
            {
                worksetIds.Add (worksetPrev.Id);
            }
        }

        OpenOptions openOpts = new OpenOptions();
        WorksetConfiguration openConfig = new WorksetConfiguration();
        // Set to close worksets by default
        openConfig.CloseAll();
        // Set list of worksets for opening
        openConfig.Open (worksetIds);
        openOpts.SetOpenWorksetsConfiguration (openConfig);
        Document doc = app.OpenDocumentFile (projectPath, openOpts);
    }
    catch (Exception e)
    {
        TaskDialog.Show ("Open File Failed", e.Message);
    }
}
```

**5.13.4 可见性和显示 (Visibility and Display)**

使用 `View.SetWorksetVisibility()` 可为特定视图设置工作集的可见性。`WorksetVisibility` 选项可以是可见的 (如果工作集是打开的, 则它将可见)、隐藏的及 `UseGlobalSetting` (指示不可覆盖该视图的设置)。相应的 `View.GetWorksetVisibility()` 方法检索该视图中工作集的当前可见性设置。然而, 此方法并未考虑工作集当前是否已打开。要确定视图中工作集是否可见, 包括考虑工作集是打开的还是关闭的, 请使用 `View.IsWorksetVisible()`。

类 `WorksetDefaultVisibilitySettings` 管理文件中工作集的默认可见性。它并不适用于族文件。如果未启用文件工作共享, 则所有图元被移到单个工作集; 只要工作共享 (重新) 启用, 则不管任何当前设置, 该工作集以及任何 (重) 创建的工作集都默认是可见的。

代码 5-44 演示了如何隐藏一个给定视图的工作集, 并根据默认设置在其他视图中隐藏。



### 代码 5-44: 隐藏工作集

```
public void HideWorkset (Document doc, View view, WorksetId worksetId)
{
    // get the current visibility
    WorksetVisibility visibility = view.GetWorksetVisibility (worksetId);

    // and set it to 'Hidden' if it is not hidden yet
    if (visibility != WorksetVisibility.Hidden)
    {
        view.SetWorksetVisibility (worksetId, WorksetVisibility.Hidden);
    }

    // Get the workset's default visibility
    WorksetDefaultVisibilitySettings defaultVisibility = WorksetDefaultVisibilitySettings.GetWorksetDefaultVisibilitySettings
(doc);

    // and making sure it is set to 'false'
    if (true == defaultVisibility.IsWorksetVisibile (worksetId))
    {
        defaultVisibility.SetWorksetVisibility (worksetId, false);
    }
}
```

除了获取和设置工作集可见性相关信息以外, 视图类还提供方法来访问工作共享显示模式和设置的信息 (代码 5-45)。WorksharingDisplayMode 枚举指示视图的模式见表 5-8。

表 5-8

视图的模式

成 员 名 称	说 明
Off	无活动的工作共享显示模式
CheckoutStatus	视图显示图元的检出状态
Owners	视图显示指定的图元所有者
ModelUpdates	视图显示模式更新
Worksets	视图显示各个图元分配到哪个工作集

### 代码 5-45: 工作共享显示模式

```
public void GetWorksharingDisplayMode (View view)
{
    // Get and Set worksharingDisplayMode of a given view
    WorksharingDisplayMode displayMode = view.GetWorksharingDisplayMode();
    Autodesk.Revit.UI.TaskDialog.Show ("GetWorksharingDisplayMode", "WorksharingDisplayMode : " + displayMode);

    view.SetWorksharingDisplayMode (WorksharingDisplayMode.CheckoutStatus);
    Autodesk.Revit.UI.TaskDialog.Show ("SetWorksharingDisplayMode", "CheckoutStatus was set for View: " + view.Name);
}
```

当图元以任意工作共享显示模式显示时, WorksharingDisplaySettings 类可控制它们如何显示 (代码 5-46)。存储在这些设置的颜色是一种常见的设置, 模型中的所有用户都可



以共享。某一个给定的颜色被应用于特定的当前用户时，其他用户将无法共享。需要注意的是，即使在非工作共享模型中，这些设置也是可用的。这是为了让工作集启用之前可以预先配置显示设置，以便可以将它们存储在样板文件中。

**代码 5-46：工作共享显示图形设置**

```
public WorksharingDisplayGraphicSettings GetWorksharingDisplaySettings (Document doc, String userName, WorksetId
worksetId, bool ownedbyCurrentUser)
{
    WorksharingDisplayGraphicSettings graphicSettings;

    // get or create a WorksharingDisplaySettings current active document
    WorksharingDisplaySettings displaySettings = WorksharingDisplaySettings.GetOrCreateWorksharingDisplaySettings (doc);

    // get graphic settings for a user, if specified
    if (!String.IsNullOrEmpty (userName))
        graphicSettings = displaySettings.GetGraphicOverrides (userName);

    // get graphicSettings for a workset, if specified
    else if (worksetId != WorksetId.InvalidWorksetId)
        graphicSettings = displaySettings.GetGraphicOverrides (worksetId);

    // get graphic settings for the OwnedByCurrentUser status
    else if (ownedbyCurrentUser)
        graphicSettings = displaySettings.GetGraphicOverrides (CheckoutStatus.OwnedByCurrentUser);

    // otherwise get graphic settings for the CurrentWithCentral status
    else
        graphicSettings = displaySettings.GetGraphicOverrides (ModelUpdatesStatus.CurrentWithCentral);

    return graphicSettings;
}
```

重载方法 `WorksharingDisplaySettings.SetGraphicOverrides()`，会基于给定的条件来设置分配给图元的图形覆盖（代码 5-47）。`WorksharingDisplaySettings` 类还可用于控制在显示的文件用户中列出哪些用户。

**代码 5-47：图形覆盖**

```
public void SetWorksharingDisplaySettings (Document doc, WorksetId worksetId, String userName)
{
    String message = String.Empty;

    // get or create a WorksharingDisplaySettings current active document
    WorksharingDisplaySettings displaySettings = WorksharingDisplaySettings.GetOrCreateWorksharingDisplaySettings (doc);

    // set a new graphicSettings for CheckoutStatus - NotOwned
    WorksharingDisplayGraphicSettings graphicSettings = new WorksharingDisplayGraphicSettings (true, new Color (255, 0,
0));
    displaySettings.SetGraphicOverrides (CheckoutStatus.NotOwned, graphicSettings);
}
```



```
// set a new graphicSettings for ModelUpdatesStatus - CurrentWithCentral
graphicSettings = new WorksharingDisplayGraphicSettings (true, new Color (128, 128, 0));
displaySettings.SetGraphicOverrides (ModelUpdatesStatus.CurrentWithCentral, graphicSettings);

// set a new graphicSettings by a given userName
graphicSettings = new WorksharingDisplayGraphicSettings (true, new Color (0, 255, 0));
displaySettings.SetGraphicOverrides (userName, graphicSettings);

// set a new graphicSettings by a given workset Id
graphicSettings = new WorksharingDisplayGraphicSettings (true, new Color (0, 0, 255));
displaySettings.SetGraphicOverrides (worksetId, graphicSettings);
}
```

**RemoveUsers()**方法从显示的用户列表中删除用户并永久丢弃其任何图形定制 (代码 5-48)。仅当用户不拥有文件中的任何图元时才可以删除。**RestoreUsers()**方法将已删除的用户添加到显示用户列表, 并许可这些用户的图形定制。请注意, 所有被还原的用户将以默认的图形覆盖显示, 之前存在的随该用户删除的任何自定义将无法恢复。

#### 代码 5-48: 删除用户

```
public void RemoveAndRestoreUsers (Document doc)
{
    // get or create a WorksharingDisplaySettings current active document
    WorksharingDisplaySettings displaySettings = WorksharingDisplaySettings.GetOrCreateWorksharingDisplaySettings (doc);

    // get all users with GraphicOverrides
    ICollection<string> users = displaySettings.GetAllUsersWithGraphicOverrides ();

    // remove the users from the display settings (they will not have graphic overrides anymore)
    ICollection<string> outUserList;
    displaySettings.RemoveUsers (doc, users, out outUserList);

    // show the current list of removed users
    ICollection<string> removedUsers = displaySettings.GetRemovedUsers ();

    String message = "Current list of removed users: ";
    if (removedUsers.Count > 0 )
    {
        foreach (String user in removedUsers)
        {
            message += "\n" + user;
        }
    }
    else
    {
        message = " [Empty] ";
    }

    TaskDialog.Show ("Users Removed", message);
}
```





```
// restore the previously removed users
int number = displaySettings.RestoreUsers (outUserList);

// again, show the current list of removed users
// it should not contain the users that were restored
removedUsers = displaySettings.GetRemovedUsers( );

message = "Current list of removed users: ";
if (removedUsers.Count > 0 )
{
    foreach (String user in removedUsers)
    {
        message += "\n" + user;
    }
}
else
{
    message = " [Empty] ";
}

TaskDialog.Show ("Removed Users Restored", message);
}
```

### 5.13.5 工作集 (Worksets)

工作集是将 Revit 文件中的图元集划分为工作共享子集的一种方式。一个文件中可以有一个或多个工作集。

文件包含一个 **WorksetTable**，这个表格包含对该文件所含的所有工作集的引用。每个文件都有一个 **WorksetTable**。表中至少有一个默认的工作集，即使文件中工作共享尚未启用。**Document.IsWorkshared** 属性可用于确定文件中工作共享是否已启用。代码 5-49 演示了用 **GetActiveWorkset** 获得文件中的 **WorksetTable**。

#### 代码 5-49：获取活动工作集

```
public Workset GetActiveWorkset (Document doc)
{
    // Get the workset table from the document
    WorksetTable worksetTable = doc.GetWorksetTable( );
    // Get the Id of the active workset
    WorksetId activeId = worksetTable.GetActiveWorksetId( );
    // Find the workset with that Id
    Workset workset = worksetTable.GetWorkset (activeId);
    return workset;
}
```

**FilteredWorksetCollector** 用于搜索、过滤和遍历一组工作集。可以指定返回工作集的过滤条件。如果没有指定条件，过滤器则会访问文件中的所有工作集。**WorksetKind** 枚举器适用于过滤工作集，见代码 5-50。

**代码 5-50: 过滤工作集**

```

public void GetWorksetsInfo (Document doc)
{
    String message = String.Empty;
    // Enumerating worksets in a document and getting basic information for each
    FilteredWorksetCollector collector = new FilteredWorksetCollector (doc);

    // find all user worksets
    collector.OfKind (WorksetKind.UserWorkset);
    IList<Workset> worksets = collector.ToWorksets();

    // get information for each workset
    int count = 3; // show info for 3 worksets only
    foreach (Workset workset in worksets)
    {
        message += "Workset : " + workset.Name;
        message += "\nUnique Id : " + workset.UniqueId;
        message += "\nOwner : " + workset.Owner;
        message += "\nKind : " + workset.Kind;
        message += "\nIs default : " + workset.IsDefaultWorkset;
        message += "\nIs editable : " + workset.IsEditable;
        message += "\nIs open : " + workset.IsOpen;
        message += "\nIs visible by default : " + workset.IsVisibleByDefault;

        TaskDialog.Show ("GetWorksetsInfo", message);

        if (0 == --count)
            break;
    }
}

```

如上例所示, 工作集类提供了许多属性来获取给定的工作集, 如所有者及工作集是否可编辑。

**5.13.6 工作共享文件管理 (Workshared File Management)**

有几个 Document 方法可用于工作共享项目文件。

**1. 启用工作共享 (Enable Worksharing)**

如果文件尚未工作共享, 可由 Document.IsWorkshared 属性来确定, 可以通过 Revit API 使用 Document.EnableWorksharing worksharing() 方法来启用工作共享。此命令将清除文件的撤销历史记录, 因此该命令以及在此之前执行的其他操作无法撤销。此外, 显式启动的所有事务阶段 (如事务、事务组和子事务) 在调用 EnableWorksharing() 之前务必完成。

**2. 重新载入最新更改 (Reload Latest)**

方法 Document.ReloadLatest() 可从中心模型中检索由于一次或多次中心同步所做的更改, 并将它们合并到当前会话。

代码 5-51 演示了使用 ReloadLatest() 来更新当前会话, 然后调用 Document.HasAllChangesFromCentral() 以确认在 ReloadLatest 执行期间没有新的“与中心同步”执行。

**代码 5-51: 从中心重新载入**

```
void ReloadDocument (Document doc)
{
    ReloadLatestOptions reloadOpts = new ReloadLatestOptions();
    try
    {
        doc.ReloadLatest (reloadOpts);
        // Check to make sure no new changes were synced with Central during reload
        if (doc.HasAllChangesFromCentral() == false)
        {
            // If there are changes from central, reload again
            doc.ReloadLatest (reloadOpts);
        }
    }
    catch (Exception e)
    {
        TaskDialog.Show ("Reload Failed", e.Message);
    }
}
```

**3. 与中心模型同步 (Synchronizing with Central Model)**

方法 `Document.SynchronizeWithCentral()` 重新载入来自中心模型的任何更改, 以使当前会话是最新的, 然后将本地更改保存回中心。即使并未作更改, “保存到中心” 也会执行。

在使用 `SynchronizeWithCentral()` 时, 选项可被指定为访问中心模型以及与其同步。访问中心的主要选项是, 确定如果模型被锁定时调用如何进行。因同步需要临时锁定中心模型, 若模型已被锁定则无法执行。默认行为是等待并多次尝试锁定中心模型以进行同步。使用 `SynchronizeWithCentral()` 方法的 `TransactWithCentralOptions` 参数, 可以重写此行为。

此方法的 `SynchronizeWithCentralOptions` 参数用于设置实际的同步选项, 如同步期间是否应该放弃当前用户所拥有的图元或工作集。

代码 5-52 演示了与中央模型同步。如果中心模型已被锁定, 则立即放弃。

**代码 5-52: 与中心同步**

```
public void SyncWithoutRelinquishing (Document doc)
{
    // Set options for accessing central model
    TransactWithCentralOptions transOpts = new TransactWithCentralOptions();
    SynchLockCallback transCallBack = new SynchLockCallback();
    // Override default behavior of waiting to try again if the central model is locked
    transOpts.SetLockCallback (transCallBack);

    // Set options for synchronizing with central
    SynchronizeWithCentralOptions syncOpts = new SynchronizeWithCentralOptions();
    // Sync without relinquishing any checked out elements or worksets
    RelinquishOptions relinquishOpts = new RelinquishOptions (false);
    syncOpts.SetRelinquishOptions (relinquishOpts);
    // Do not automatically save local model after sync
}
```



```

syncOpts.SaveLocalAfter = false;
syncOpts.Comment = "Changes to Workset!";

try
{
    doc.SynchronizeWithCentral (transOpts, syncOpts);
}
catch (Exception e)
{
    TaskDialog.Show ("Synchronize Failed", e.Message);
}
}

class SynchLockCallback : ICentralLockedCallback
{
    // If unable to lock central, give up rather than waiting
    public bool ShouldWaitForLockAvailability()
    {
        return false;
    }
}

```

#### 4. 创建新的本地模型 (Create New Local Model)

WorksharingUtils.CreateNewLocal()方法复制一个中心模型到新的本地文件。此方法不会打开新文件。

## 5.14 构造建模 (Construction Modeling)

Revit API 允许将图元分割为子零件或将它们收集到部件中, 以支持构造建模 workflow, 与 Revit 用户界面所能做的差不多。零件和部件都可以独立列出明细、标记、过滤和输出, 还可以将零件分割为更小的零件。在创建部件类型之后, 可以将其附加的实例放置到项目中, 并生成独立的部件视图。

构造建模的主要类有:

- **AssemblyInstance:** 此类将多个图元合并, 以便标记、过滤、列出明细及创建独立的组件视图。
- **AssemblyType:** 表示构造部件图元的类型。项目中每个新创建的独特部件自动建立相应的部件类型。根据现有部件类型, 可将新部件实例放置于文件中。
- **PartUtils:** 此实用程序类包含常用零件的实用方法, 包括能够创建零件、分割零件, 并获取有关零件信息。
- **AssemblyViewUtils:** 一个创建各种类型的部件视图的实用程序类。

### 5.14.1 部件和视图 (Assemblies and Views)

#### 1. 部件 (Assemblies)

部件实例类的静态方法 Create()用于在项目中创建一个新的部件实例。Create()方法必



须在事务内创建，并且必须在新创建的部件实例上执行任何操作之前提交事务。在事务完成之后，部件类型才被指定。

代码 5-53 演示了创建一个新部件实例，更改其部件类型名称，然后创建部件实例的一些视图。

#### 代码 5-53: 创建部件和视图

```
void CreateAssemblyAndViews (Autodesk.Revit.DB.Document doc, ICollection<ElementId> elementIds)
{
    Transaction transaction = new Transaction (doc);
    // use category of one of the assembly elements
    ElementId categoryId = doc.GetElement (elementIds.First()).Category.Id;
    if (AssemblyInstance.IsValidNamingCategory (doc, categoryId, elementIds))
    {
        transaction.Start ("Create Assembly Instance");
        AssemblyInstance assemblyInstance = AssemblyInstance.Create (doc, elementIds, categoryId);
        // commit the transaction that creates the assembly instance before modifying the instance's name
        transaction.Commit();

        transaction.Start ("Set Assembly Name");
        assemblyInstance.AssemblyTypeName = "My Assembly Name";
        transaction.Commit();

        // create assembly views for this assembly instance
        if (assemblyInstance.AllowsAssemblyViewCreation())
        {
            transaction.Start ("View Creation");
            View3D view3d = AssemblyViewUtils.Create3DOrthographic (doc, assemblyInstance.Id);
            ViewSchedule partList = AssemblyViewUtils.CreatePartList (doc, assemblyInstance.Id);
            transaction.Commit();
        }
    }
}
```

创建部件实例的另一个方法是使用现有部件类型。要使用现有部件类型来创建部件实例，请使用静态方法 `AssemblyInstance.PlaceInstance()` 并指定要用到的部件类型的图元 ID，以及用来放置部件的位置。

#### 2. 部件视图 (Assembly Views)

使用 `AssemblyViewUtils` 类，可以创建部件实例的各种部件视图，包括正交三维部件视图、详图剖面部件视图，材料提取多类别明细表部件视图、零件清单多类别明细表部件视图以及图纸部件视图。在使用任何这些新创建的部件视图之前，文件必须重新生成。通过上面的示例可发现，创建两个新部件视图后，事务被提交，`Commit()`方法会自动重新生成文件。

#### 5.14.2 零件 (Parts)

零件可由具有层状结构的图元生成，例如：

- 墙（不包括叠层墙和幕墙）。
- 楼板（不包括形状编辑楼板）。



- 屋顶（不包括脊线屋顶）。
- 天花板。
- 结构板基础。

在 Revit API 中, 可使用 `PartUtils` 类将图元分割为零件。静态方法 `PartUtils.CreateParts()` 用于从一个或多个图元来创建零件。需要注意的是, 与 API 中的大多数图元的创建方法不同, `CreateParts()` 实际上并不创建或恢复零件, 更确切地说是实例化一个名为 `PartMaker` 的图元。`PartMaker` 使用其内嵌规则在重生成期间来驱动创建所需的零件。

API 还提供了一个接口 `PartUtils.DivideParts()` 来细分零件。作为该接口的输入, 可接受零件 ID 的集合、“相交图元” ID (可以是层和网格) 的集合和曲线的集合。该程序使用相交的图元和曲线作为边界, 从中分割并生成新的部件。

调用 `GetAssociatedParts()` 方法可以找到一些或所有与某个图元关联的零件, 或者使用 `HasAssociatedParts()` 来确定某图元是否有零件。

可以通过 API 删除零件, 删除个别零件图元, 也可以删除关联到零件的 `PartMaker` (这会在下次重生成后删除由此 `PartMaker` 生成的所有零件)。

Revit API 中对零件的操控和 Revit 用户界面中大致相同。例如, 用 `PartUtils.SetFace Offset()` 可以偏移零件的外边界。

代码 5-54 演示了偏移可被偏移的零件的所有面。图 5-18 为所选零件的偏移前后的面。

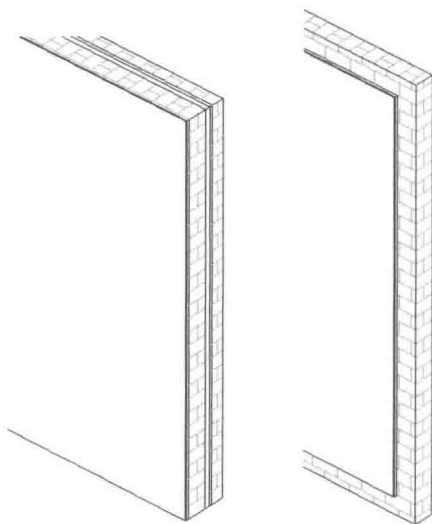


图 5-18 所选零件的偏移前后的面

#### 代码 5-54: 偏移零件的面

```
Part part = doc.GetElement (uidoc.Selection.PickObject (ObjectType.Element)) as Part;
Autodesk.Revit.DB.GeometryElement geomElem = part.get_Geometry (newOptions());

foreach (GeometryObject geomObject in geomElem)
{
    if (geomObject is Solid)
    {
        Solid solid = geomObject as Solid;
        FaceArray faceArray = solid.Faces;
        foreach (Face face in faceArray)
        {
            if (part.CanOffsetFace (face))
            {
                part.SetFaceOffset (face, 1);
            }
        }
    }
}
```



## 5.15 链接文件 (Linked Files)

Revit API 可以确定哪些 Revit 图元引用外部文件 (“链接的文件”), 并可以对 Revit 如何加载外部文件做一些修改。

包含外部文件引用的图元是这样图元, 它引用 “基础” .rvt 文件以外的某些文件。例子包括 Revit 链接、CAD 链接、存储 “基” 文件位置的图元, 及渲染贴花。判断图元有无外部文件, 请使用 `Element.IsExternalFileReference()`。 `Element.GetExternalFileReference()` 返回给定图元的 `ExternalFileReference`, 其中包含图元引用外部文件有关的信息。

下面是与 Revit API 中链接文件有关的类:

- `ExternalFileReference`: 非图元类, 其中包含 Revit 项目所引用的单个外部文件路径和类型信息。
- `ExternalFileUtils`: 实用程序类, 它允许用户查找所有的外部文件引用, 根据某个图元获得外部文件引用, 或者判断图元有无外部文件引用。
- `RevitLinkType`: 图元, 表示 Revit 文件链接到 Revit 项目。
- `ModelPath`: 非图元类, 其中包含文件的路径信息 (不一定是 .rvt 文件)。路径可以指向本地驱动器或网络驱动器位置, 或者 Revit 服务器位置。
- `ModelPathUtils`: 实用程序类, 提供字符串与模型路径之间转换的方法。
- `TransmissionData`: 该类在文件中存储所有外部文件引用有关信息。Revit 项目的 `TransmissionData` 无需打开文件就可读取。

模型路径 (`ModelPath`) 是指向另一个文件的路径, 可以引用 Revit 模型, 或任何 Revit 的外部文件引用, 如 DWG 链接。路径可以是相对路径也可以是绝对路径, 但它们必须包含一个指明文件类型的扩展。相对路径通常是相对于当前打开的文件。如果当前文件是工作共享的, 则路径被视为相对于中心模型。要创建模型路径, 请使用 `FilePath` 或 `ServerPath` 两者之一的派生类。

`ModelPathUtils` 类包含模型路径与用户可见的路径字符串之间互相转换的实用函数, 以及确定一个字符串是否有效的服务器路径。

### 5.15.1 Revit 链接 (Revit Links)

Revit 文件可以链接到各种外部文件, 包括其他 Revit 文件。在 Revit API 中, 这些类型的链接是由 `RevitLinkType` 和 `RevitLinkInstance` 类来表示。 `RevitLinkType` 类表示另一个 Revit 文件 “链接” 纳入当前的 “主体”, 而 `RevitLinkInstance` 类表示 `RevitLinkType` 的一个实例。

#### 1. 创建新链接 (Creating New Links)

要创建一个新的 Revit 链接, 请使用静态方法 `RevitLinkType.Create()` 创建一个新的 Revit 链接类型并载入链接文件, 并使用静态方法 `RevitLinkInstance.Create()` 在模型中放置链接实例。静态方法 `RevitLinkType.Create()` 需要一个文件 (主体文件)、一个被链接的文件的模型路径和一个 `RevitLinkOptions` 对象。 `RevitLinkOptions` 类表示创建和载入 Revit 链



接的选项。选项包括 Revit 是否要存储链接文件的相对路径或绝对路径和工作集的配置。创建链接, 打开工作集后, 需要设定 `WorksetConfiguration` 类。需要注意的是, 相对或绝对路径决定了 Revit 将如何存储路径, 但传递给 `Create()` 方法的模型路径需能找到链接的初始文件的完整路径。

代码 5-55 演示了如何使用 `RevitLinkType.Create()` 创建新的链接, 其中变量 `pathName` 是硬盘上被链接文件的完整路径。

#### 代码 5-55: 创建新的 Revit 链接

```
public ElementId CreateRevitLink (Document doc, string pathName)
{
    FilePath path = new FilePath (pathName);
    RevitLinkOptions options = new RevitLinkOptions (false);
    // Create new revit link storing absolute path to file
    RevitLinkLoadResult result = RevitLinkType.Create (doc, path, options);
    return (result.ElementId);
}
```

创建 `RevitLinkType` 之后, 即可将实例添加到文件中。在代码 5-56 中, 添加了 `RevitLinkType` 的两个实例, 原位实例和偏移 100 英尺后的实例。创建 `RevitLinkInstance` 之前, Revit 链接会显示在管理链接窗口中, 但任何视图中, 链接的文件的图元是不可见的。

#### 代码 5-56: 新建 Revit 链接实例

```
public void CreateLinkInstances (Document doc, ElementId linkTypeId)
{
    // Create revit link instance at origin
    RevitLinkInstance.Create (doc, linkTypeId);
    RevitLinkInstance instance2 = RevitLinkInstance.Create (doc, linkTypeId);
    // Offset second instance by 100 feet
    Location location = instance2.Location;
    location.Move (new XYZ (0, -100, 0));
}
```

代码 5-57 演示了如何获取 `WorksetConfiguration` 并修改, 以保证仅打开一个指定工作集, 并将 `RevitLinkOptions` 恢复到原先设置, 以创建新的链接。

#### 代码 5-57: 创建有一个打开工作集的链接

```
public bool CreateRevitLinkWithOneWorksetOpen (Document doc, string pathName, string worksetName)
{
    FilePath path = new FilePath (pathName);
    RevitLinkOptions options = new RevitLinkOptions (true);

    // Get info on all the user worksets in the project prior to opening
    IList<WorksetPreview> worksets = WorksharingUtils.GetUserWorksetInfo (path);
    IList<WorksetId> worksetIds = new List<WorksetId>();
    // Find worksetName
    foreach (WorksetPreview worksetPrev in worksets)
```





```
{
    if (worksetPrev.Name.CompareTo (worksetName) == 0)
    {
        worksetIds.Add (worksetPrev.Id);
        break;
    }
}

// close all worksets but the one specified
WorksetConfiguration worksetConfig = options.GetWorksetConfiguration ();
worksetConfig.CloseAll ();
worksetConfig.Open (worksetIds);
options.SetWorksetConfiguration (worksetConfig);

RevitLinkLoadResult result = RevitLinkType.Create (doc, path, options);
RevitLinkType type = doc.GetElement (result.ElementId) as RevitLinkType;
return (result.LoadResult == RevitLinkLoadResultType.LinkLoaded);
}
```

是否创建或载入一个链接，可根据返回的 `RevitLinkLoadResults` 予以确认。该类具有一个属性来确定链接是否已载入。它还有一个 `ElementId` 属性，表示所创建或载入的链接模型的 ID。

## 2. 载入和卸载链接 (Loading and Unloading Links)

`RevitLinkType` 有几个与载入链接相关的方法。`Load()`、`LoadFrom()`和 `Unload()`允许载入、卸载，或从新位置载入一个链接。这些方法会重生成文件，清除文件的撤销历史记录。在调用这些方法之前，所有显式启动的事务阶段（如事务、事务组和子事务）必须完成。

静态方法 `RevitLinkType.IsLoaded()`将返回链接是否被加载。

## 3. 获取链接信息 (Getting Link Information)

文件中每个 `RevitLinkType` 可以有一个或多个关联的 `RevitLinkInstances`。`RevitLinkInstance.GetLinkDocument` 方法返回一个与 `Revit` 链接关联的文件。此文件不能修改，这意味着无法执行需要事务的操作或在内存中修改文件状态（如保存和关闭）。

可以使用从 `ElementId.GetTypeId()`方法所获得的 `ElementId`，来从文件中检索 `RevitLinkInstance` 关联的 `RevitLinkType`。链接文件的 `RevitLinkType` 有几个嵌套链接的相关方法和属性。链接到其他文件的文件本身可能有链接。如果 `RevitLinkType` 是一个嵌套链接（即它有一些其他链接作为父链接），则 `IsNested` 属性返回“true”，或若它是一个顶级链接则返回“false”。方法 `GetParentId()`会获取此链接的直接父链接 ID，而 `GetRootId()`将返回顶级链接 ID，它是此链接的“根”。如果链接是顶级链接，则这两种方法都将返回“invalidElementId”。`GetChildIds()`方法将返回直接链接到该链接的所有链接的图元 ID。

例如，如果 C 链接到文件 B，B 转而链接到文件 A，那么对 C 链接调用 `GetParentId()`将返回文件 B 的 ID，对 C 链接调用 `GetRootId()`将返回文件 A 的 ID。对文件 A 调用 `GetChildIds()`将只返回 B 链接的 ID，因为 C 并不直接链接到 A。

`RevitLinkType` 还有一个 `PathType` 属性，指示外部文件引用的路径是否主体文件位置的相对路径（或中心模型的位置，若主体是工作共享的）、硬盘或网络位置的绝对路径，或



者是否一个指向 Revit 服务器位置的路径。

RevitLinkType 的 AttachmentType 属性指示链接是一个“附件”还是“叠加”。“附件”链接被认为是其父链接的一部分且会一并带上，如果它们的父链接又链接到另一个文件。当直接打开父链接时，“叠加”链接是唯一可见的。

代码 5-58 演示了如何获取文件中的所有 RevitLinkInstances 并显示一些与其有关的信息。

**代码 5-58: 获取链接信息**

```
public void GetAllRevitLinkInstances (Document doc)
{
    FilteredElementCollector collector = new FilteredElementCollector (doc);
    collector.OfClass (typeof (RevitLinkInstance));
    StringBuilder linkedDocs = new StringBuilder();
    foreach (Element elem in collector)
    {
        RevitLinkInstance instance = elem as RevitLinkInstance;
        Document linkDoc = instance.GetLinkDocument();
        linkedDocs.AppendLine ("FileName: " + Path.GetFileName (linkDoc.PathName));
        RevitLinkType type = doc.GetElement (instance.GetTypeId()) as RevitLinkType;
        linkedDocs.AppendLine ("Is Nested: " + type.IsNestedLink.ToString());
    }

    TaskDialog.Show ("Revit Links in Document", linkedDocs.ToString());
}
```

#### 4. 链接几何图形 (Link Geometry)

(1) 共享坐标 (Shared Coordinates)。RevitLinkType 方法 SavePositions() 和 HasSaveablePositions() 支持将共享的坐标更改保存回所链接的文件。使用 HasSaveablePositions() 以确定链接是否有共享的定位变化可以保存。调用 SavePositions() 以将共享的坐标更改保存回所链接的文件。SavePositions() 需要一个 ISaveSharedCoordinatesCallback 接口的实例以解决 Revit 遇到被修改的链接时的状况。该接口的 GetSaveModifiedLinksOption() 方法确定 Revit 是否该保存链接、不保存链接，或者完全放弃共享定位。

虽然 SavePositions() 不清除文件的撤销历史记录，但它仍然无法撤销，因为它将链接的共享坐标更改保存到了硬盘上。

(2) 转换几何参照 (Conversion of Geometric References)。Reference 类拥有相关链接文件的成员，允许在仅参照链接内容的 Reference 对象和参照主体的 Reference 对象之间进行转换。这允许应用程序基于链接中的几何图元，找到所需的面，并将主体中的面转换成参照平面，作为适合放置一个基于面的实例使用的参照。此外，这些 Reference 成员使得有可能在主体中获得一个参照(例如从尺寸或族)，并将其转换为适合 Element.GetGeometryObjectFromReference() 使用的链接中的参照。

Reference.LinkElementId 属性表示链接文件中被此参照所引用的顶级图元的 ID，或 InvalidElementId 对未引用链接的 RVT 文件中的图元的参照。Reference.CreateLinkReference() 方法使用 RevitLinkInstance，从 Revit 链接中的 Reference 创建一个 Reference。而 Reference.



CreateReferenceInLink()方法从主体文件中的 Reference 创建一个 Revit 链接中的 Reference。

(3) 选择链接中的图元 (Picking Elements in Links)。选择方法 PickObject() 和 PickObjects() 允许在 Revit 链接中选择对象。要让用户选择链接文件中的图元, 请使用 ObjectType.LinkedElement 枚举值作为 PickObject() 或 PickObjects() 的第一个参数。请注意, 此选项只允许在链接中选择图元, 而不是在主体文件中。

代码 5-59 演示了使用一个 ISelectionFilter, 只允许在链接文件中选择墙。

代码 5-59: 在链接文件中选择图元

```
public void SelectElementsInLinkedDoc (Autodesk.Revit.DB.Document document)
{
    UIDocument uidoc = new UIDocument (document);
    Selection choices = uidoc.Selection;
    // Pick one wall from Revit link
    WallInLinkSelectionFilter wallFilter = new WallInLinkSelectionFilter ();
    Reference elementRef = choices.PickObject (ObjectType.LinkedElement, wallFilter, "Select a wall in a linked document");
    if (elementRef != null)
    {
        TaskDialog.Show ("Revit", "Element from link document selected.");
    }
}

// This filter allows selection of only a certain element type in a link instance.
class WallInLinkSelectionFilter : ISelectionFilter
{
    private RevitLinkInstance m_currentInstance = null;

    public bool AllowElement (Element e)
    {
        // Accept any link instance, and save the handle for use in AllowReference()
        m_currentInstance = e as RevitLinkInstance;
        return (m_currentInstance != null);
    }

    public bool AllowReference (Reference refer, XYZ point)
    {
        if (m_currentInstance == null)
            return false;

        // Get the handle to the element in the link
        Document linkedDoc = m_currentInstance.GetLinkDocument();
        Element elem = linkedDoc.GetElement (refer.LinkedElementId);

        // Accept the selection if the element exists and is of the correct type
        return elem != null && elem is Wall;
    }
}
```



### 5.15.2 管理外部文件 (Managing External Files)

#### 1. 外部文件实用工具 (ExternalFileUtils)

顾名思义, 这个工具类提供有关外部文件引用的信息。ExternalFileUtils.ElementIds.GetAllExternalFileReferences()方法返回一个集合, 包含文件中外部文件引用的所有图元的图元 ID。(请注意, 它不会返回嵌套 Revit 链接的 ID; 而只返回顶级的引用。)此实用工具类有另外两种方法, IsExternalFileReference()和 GetExternalFileReference()执行类似 Element 类的命名方法相同的功能, 但只用于已知一个 ElementId 的情况, 而不是第一个获取图元。

#### 2. 数据传输 (TransmissionData)

TransmissionData 存储外部文件引用的先前状态及被请求状态的有关信息。这意味着它从最近一次该 TransmissionData 的文件被打开时, 开始存储加载状态和引用的路径。它还为 Revit 下次打开文件时做好准备, 存储加载状态和路径信息。

因此, 不必打开完整的 Revit 相关文件, 就可用 TransmissionData 执行外部文件引用操作。ReadTransmissionData()和 WriteTransmissionData()方法可用于获取有关外部引用的信息, 或更改信息。例如, 用已将所有引用设置为 LinkedFileStatus.Unloaded 的 TransmissionData 对象来调用 WriteTransmissionData(), 会导致下一次打开该文件不加载任何引用。

TransmissionData 无法添加或移除对外部文件的引用。如果调用 AddExternalFileReference 时所使用的 ElementId 与外部文件引用图元不对应, 则文件加载会忽略该信息。

代码 5-60 演示了对给定位置的文件读取 TransmissionData, 并设置所有 Revit 链接在下次打开该文件时被卸载。

#### 代码 5-60: 卸载 Revit 链接

```
void UnloadRevitLinks (ModelPath location)
/// This method will set all Revit links to be unloaded the next time the document at the given location is opened.
/// The TransmissionData for a given document only contains top-level Revit links, not nested links.
/// However, nested links will be unloaded if their parent links are unloaded, so this function only needs to look at the document's
immediate links.
{
    // access transmission data in the given Revit file
    TransmissionData transData = TransmissionData.ReadTransmissionData (location);
    if (transData != null)
    {
        // collect all (immediate) external references in the model
        ICollection<ElementId> externalReferences = transData.GetAllExternalFileReferenceIds();
        // find every reference that is a link
        foreach (ElementId refId in externalReferences)
        {
            ExternalFileReference extRef = transData.GetLastSavedReferenceData (refId);
            if (extRef.ExternalFileReferenceType == ExternalFileReferenceType.RevitLink)
            {
                // we do not want to change neither the path nor the path-type
                // we only want the links to be unloaded (shouldLoad = false)
                transData.SetDesiredReferenceData (refId, extRef.GetPath(), extRef.PathType, false);
            }
        }
    }
}
```



```

    }
}

// make sure the IsTransmitted property is set
transData.IsTransmitted = true;
// modified transmission data must be saved back to the model
TransmissionData.WriteTransmissionData (location, transData);
}
else
{
    Autodesk.Revit.UI.TaskDialog.Show ("Unload Links", "The document does not have any transmission data");
}
}

```

### 3. 构建服务器位置的模型路径 (Construct ModelPath for Location on Server)

要读取 TransmissionData 对象, 需要调用静态方法 TransmissionData.ReadTransmissionData()。它需要一个 ModelPath 对象。

有两种构建引用中心文件的 ModelPath 对象的方法。第一种方法涉及使用 ModelPathUtils 和 ModelPath 基类, 步骤如下:

(1) 组成用户可见的中心文件路径字符串: ModelPathUtils.GetRevitServerPrefix()+“相对路径”。

注意: 用于“相对路径”的文件夹分隔符是个正斜杠 (/)。通过 ModelPathUtils.ConvertUserVisiblePathToModelPath()方法创建一个 ModelPath 对象。传入上一步所组成的字符串。

(2) 通过 TransmissionData.: ReadTransmissionData()方法读取传输数据。传入上一步所获得的 ModelPath。

代码 5-61 演示了此方法, 假定中心文件 testmode.RVT 存储在 Revit 服务器 SHACNG035WQRP 的根文件夹中。

#### 代码 5-61: 用 ModelPath 构建中心文件路径

```

[TransactionAttribute (Autodesk.Revit.Attributes.TransactionMode.Manual)]
public class RevitCommand : IExternalCommand
{
    public Result Execute (ExternalCommandData commandData,
        ref string messages, ElementSet elements)
    {
        UIApplication app = commandData.Application;
        Document doc = app.ActiveUIDocument.Document;
        Transaction trans = new Transaction (doc, "ExComm");
        trans.Start();
        string visiblePath = ModelPathUtils.GetRevitServerPrefix() + "/testmode.rvt";
        ModelPath serverPath = ModelPathUtils.ConvertUserVisiblePathToModelPath (visiblePath);
        TransmissionData transData = TransmissionData.ReadTransmissionData (serverPath);
        string mymessage = null;
        if (transData != null)
        {

```



```

        //access the data in the transData here.
    }
    else
    {
        Autodesk.Revit.UI.TaskDialog.Show ("Unload Links",
            "The document does not have any transmission data");
    }
    trans.Commit();
    return Result.Succeeded;
}
}

```

第二种构建引用中心文件的 `ModelPath` 对象的方法是使用子类 `ServerPath`。如果程序知道本地服务器名称，则可以使用这种方法，然而，并不建议这种方法，因为服务器名称有可能会被 Revit 用户从 Revit UI 中更改。步骤如下：

(1) 使用 `ServerPath` 构造函数创建 `ServerPath` 对象：`new ServerPath ("ServerName OrServerIp", "relative path without the initial forward slash")`。

注：第一个参数是服务器名称，而不是由 `ModelPathUtils.GetRevitServerPrefix()` 返回的字符串，第二个参数不包括初始的正斜杠，文件夹分隔符是一个正斜杠 (/)。请看代码 5-62。

(2) 通过 `TransmissionData.ReadTransmissionData()` 方法读取 `TransmissionData` 对象。传入上一步中获得的 `ServerPath`。

代码 5-62 演示了用 `ServerPath` 构建服务器路径。

#### 代码 5-62：用 `ServerPath` 构建服务器路径

```

[TransactionAttribute (Autodesk.Revit.Attributes.TransactionMode.Manual)]
public class RevitCommand : IExternalCommand
{
    public Result Execute (ExternalCommandData commandData,
        ref string messages, ElementSet elements)
    {
        UIApplication app = commandData.Application;
        Document doc = app.ActiveUIDocument.Document;
        Transaction trans = new Transaction (doc, "ExComm");
        trans.Start();
        ServerPath serverPath = new ServerPath ("SHACNG035WQRP", "testmodel.rvt");
        TransmissionData transData = TransmissionData.ReadTransmissionData (serverPath);
        string mymessage = null;
        if (transData != null)
        {
            //access the data in the transData here.
        }
        else
        {
            Autodesk.Revit.UI.TaskDialog.Show ("Unload Links",
                "The document does not have any transmission data");
        }
    }
}

```



```
trans.Commit();
return Result.Succeeded;
}
}
```

5.16 导出 (Export)

Revit API 允许对 Revit 文件或其一部分，导出到其他软件使用的各种格式。Document 类有一个可重载的 Export()方法，可使用 Revit 内置导出程序启动文件导出。对于更进一步的需求，某些类型的导出，可用 Revit 插件定制，如导出 IFC 及导出 Navisworks（请注意，Navisworks 导出仅可作为插件导出程序。）

表 5-9 概括了 Document.Export()方法重载。

表 5-9 Document.Export() 方法

格式	Export()参数	注 释
gbXML	String, String, MassGBXMLExportOptions	由体量模型文件导出 gbXML 文件
gbXML	String, String, GBXMLExportOptions	导出绿色建筑 XML 格式
IFC	String, String, IFCExportOptions	导出行业标准类 (IFC) 格式文件
NWC	String, String, NavisworksExportOptions	将 Revit 项目导出为 Navisworks ".nwc" 格式。 请注意，为使用此功能，必须在 Revit 会话中注册一个兼容的 Navisworks 导出程序插件
DWF	String, String, ViewSet, DWFExportOptions	将当前视图或选定的一些视图导出为 DWF 格式
DWFX	String, String, ViewSet, DWFXExportOptions	将当前视图或选定的一些视图导出为 DWFX 格式
FBX	String, String, ViewSet, FBXExportOptions	将文件导出为 3D-Studio Max (FBX) 格式
DGN	String, String, ICollection (ElementId), DGNExportOptions	将选定的一些视图导出为 DGN 格式
DWG	String, String, ICollection (ElementId), DWGExportOptions	将选定的一些视图导出为 DWG 格式
DXF	String, String, ICollection (ElementId), DXFExportOptions	将选定的一些视图导出为 DXF 格式
SAT	String, String, ICollection (ElementId), SATExportOptions	将当前视图或选定的一些视图导出为 SAT 格式
ADSK	String, String, View3D, ViewPlan, BuildingSiteExportOptions	将文件导出为 Autodesk Civil3D® 格式

1. 导出 gbXML (Exporting to gbXML)

有两种方法导出绿色建筑 XML 格式。其一，它最后一个参数 MassGBXMLExportOptions 仅适用包含一个或多个概念体量族实例的项目。传递到此方法的 MassGBXMLExport Options 对象可以只用体量分区 ID 构建以在导出的 gbXML 中分析，或用体量分区 ID 及体



量 ID 来构建, 用作导出的 gbXML 中的着色表面。当使用体量时, 它们必须没有体量楼板或体量分区, 以免最终在 gbXML 输出中复制表面信息。

用于另一个 gbXML 的 GBXMLExportOptions 对象的导出选项, 无需指定设置, 它完全使用默认设置。

## 2. 导出 IFC (Exporting to IFC)

以 IFC 选项调用 Document.Export(), 可使用默认 Revit IFC 导出实现, 亦可使用自定义 IFC 导出程序——若已注册到 Revit 当前会话。无论哪种情况, 都是用 IFCEXportOptions 类设置导出选项, 如 Revit 是否支持当前导出 IFC 标准量, 是否允许按楼层标高分割多层墙和柱。

## 3. 导出 Navisworks (Exporting to Navisworks)

Navisworks 的 Export() 方法需要一个当前 Revit 会话注册的兼容 Navisworks 导出程序插件。如果没有已注册的兼容导出程序, 该方法将引发异常。要确定是否有已注册的 Navisworks 导出程序, 请使用 OptionalFunctionalityUtils.IsNavisworksExporterAvailable() 方法。

NavisworksExportOptions 对象可对导出 Navisworks 进行各种导出设置, 如是否将文件按层标高划分、是否导出房间几何图形。此外, NavisworksExportOptions.ExportScope 属性还可指定导出范围。默认范围为模型。其他选项包括 View 和 SelectedElements。当设置 View 时, 应相应设置 NavisworksExportOptions.ViewId 属性。此属性仅用于导出范围设置为视图的情况。当设置为 SelectedElements 时, 应调用 NavisworksExportOptions.SetSelectedElementIds() 方法, 并以要导出图元的 id 为参数。

## 4. 导出 DWF 和 DWFX (Exporting to DWF and DWFX)

使用相应的 Document.Export 重载, 可以导出 DWF 和 DWFX 文件。两种方法都有 ViewSet 参数, 表示要导出什么样的视图。ViewSet 中的所有视图必须是可打印的, 以保证导出成功。而这可以使用每个视图的 View.CanBePrinted 属性进行检查。最后一个参数是 DWFXExportOptions 或 DWFXExportOptions。DWFXExportOptions 是由 DWFXExportOptions 派生并具有完全相同的导出设置。选项包括是否导出裁剪框、图像质量、纸张格式。代码 5-63 演示了导出 DWF 文件的方法。

**代码 5-63: 导出 DWF**

```
public bool ExportViewToDWF (Document document, View view, string pathname)
{
    DWFXExportOptions dwfOptions = new DWFXExportOptions ();
    // export with crop box and area and room geometry
    dwfOptions.CropBoxVisible = true;
    dwfOptions.ExportingAreas = true;
    ViewSet views = new ViewSet ();
    views.Insert (view);
    return (document.Export (Path.GetDirectoryName (pathname),
        Path.GetFileNameWithoutExtension (pathname), views, dwfOptions));
}
```

## 5. 导出 3D-Studio Max (Exporting to 3D-Studio Max)

FBX 导出需要存在某些可选的且未必属于已安装的 Revit 中一部分的模块, 以便





`OptionalFunctionalityUtils.IsFBXExportAvailable()` 方法报告 FBX 导出功能是否可用。导出到 3D Studio Max 的 `Export()` 方法有个 `ViewSet` 参数, 表示所要导出的视图设置。只允许三维视图。 `FBXExportOptions` 参数可用于指定是否导出边界、是否使用详图等级, 以及导出视图失败时导出过程是否中止。

#### 6. 导出 CAD 格式 (Exporting to CAD Formats)

导出 DGN、DWG 和 DXF 格式的文件具有类似的导出方法和选项。

DGN、DWG 和 DXF 导出都需要存在某些可选的、且未必属于已安装的 Revit 版本中一部分的模块, 以便对应方法的 `OptionalFunctionalityUtils` 类来报告这些每个类型的导出功能是否可用。

导出 DGN、DWG 和 DXF 格式的 `Export()` 方法都有一个表示所要导出的视图的参数(作为视图 `ElementIds` 的 `ICollection`)。必须至少有一个有效的视图存在, 并且必须是可打印的, 才能导出成功。而这可以用每个视图的 `View.CanBePrinted` 属性进行检查。

这些格式的每个导出选项派生自 `BaseExportOptions`, 因此有许多共同的导出设置, 例如颜色模式或是否隐藏范围框。`BaseExportOptions` 还有一个可以返回文件的任何预定义设置的静态方法 `GetPredefinedSetupNames()`。然后, 预定义设置的名称可以被传递到从相应的选项类所获得的静态方法 `GetPredefinedOptions()`: `DWGExportOptions`、`DGNExportOptions` 或 `DXFExportOptions`。DWG 和 DXF 文件的导出选项有更多的共同选项, 因为它们共享 `ACADEExportOptions` 基类。

代码 5-64 演示了使用第一个预定义设置的名字和预定义的选项, 不做任何更改, 将活动视图 (如果是可打印的) 导出到 DGN 文件。

#### 代码 5-64: 导出 DGN

```
public bool ExportDGN (Document document, View view)
{
    bool exported = false;
    // Get predefined setups and export the first one
    IList<string> setupNames = BaseExportOptions.GetPredefinedSetupNames (document);
    if (setupNames.Count > 0)
    {
        // Get predefined options for first predefined setup
        DGNExportOptions dgnOptions = DGNExportOptions.GetPredefinedOptions (document, setupNames [0]);

        // export the active view if it is printable
        if (document.ActiveView.CanBePrinted == true)
        {
            ICollection<ElementId> views = new List<ElementId> ();
            views.Add (view.Id);
            exported = document.Export (Path.GetDirectoryName (document.PathName),
                Path.GetFileNameWithoutExtension (document.PathName), views, dgnOptions);
        }
    }

    return exported;
}
```

可以对这些文件类型的各种映射设置作修改，如通过创建相应的映射表，并将其传递到 `BaseExportOptions` 类的适当方法，修改图层映射或文本的字体映射。有关更多信息，请参阅 5.16.1 节导出表主题。

7. 导出 SAT (Exporting to SAT)

SAT 导出方法有一个表示所要导出视图的参数 (作为视图 `ElementIds` 的 `ICollection`)。必须至少存在一个有效视图，且必须是可打印的，以保证导出成功。而这可以用每个视图的 `CanBePrinted` 属性进行检查。`SATExportOptions` 对象无需指定设置，完全使用默认设置。

8. 输出土木工程设计应用程序 (Exporting to Civil Engineering Design Applications)

最后一个导出方法，以土木工程设计应用程序格式导出文件的三维视图。该方法的一个参数是 `Viewplan`，用来指定总面积平面。它将导出视图平面的全部面积，且它必须是“总建筑”面积平面。要检查是否“总建筑”面积图表，请使用 `AreaScheme.GrossBuildingArea` 属性。`BuildingSiteExportOptions` 对象允许自定义设置值，如建筑面积或总占用面积。

5.16.1 导出表 (Export Tables)

表 5-10 中列出的表类，显示了对使用 DWG 和 DGN 等各种导出格式映射的读写访问。每个类包含关键字及成对的映射数据信息。

表 5-10 导 出 表 类

类	说 明
<code>ExportLayerTable</code>	该表支持各种图层属性 (类别、名称、颜色名称) 对目标导出格式中图层名称的映射
<code>ExportLinetypeTable</code>	该表支持目标导出格式中线型名称的映射
<code>ExportPatternTable</code>	该表支持目标导出格式中填充图案名称及填充图案名称 ids 的映射
<code>ExportFontTable</code>	该表支持目标导出格式中字体名称的映射
<code>ExportLineweightTable</code>	该表支持目标导出格式中线宽名称的映射

大部分这些表可通过 `BaseExportOptions` 类 (`DGN`、`DXF` 和 `DWG` 格式的选项基类) 获得。`DGNExportOptions` 类可使用 `ExportLineweightTable`。每个表都有一个相应的 `Get` 和 `Set` 方法。要修改映射，请先获取相应的表进行更改，然后将其设回选项类。

代码 5-65 演示了先修改 `ExportLayerTable`，然后导出一个 DWG 文件。

代码 5-65：用修改过的 `ExportLayerTable` 导出 DWG

```
public bool ExportDWGModifyLayerTable (Document document, View view)
{
    bool exported = false;
    IList<string> setupNames = BaseExportOptions.GetPredefinedSetupNames (document);
    if (setupNames.Count > 0)
    {
        // Get the export options for the first predefined setup
        DWGExportOptions dwgOptions = DWGExportOptions.GetPredefinedOptions (document, setupNames [0]);

        // Get the export layer table
        ExportLayerTable layerTable = dwgOptions.GetExportLayerTable ();
```



```

// Find the first mapping for the Ceilings category
string category = "Ceilings";
ExportLayerKey targetKey = layerTable.GetKeys().First<ExportLayerKey> (layerKey => layerKey.CategoryName ==
category);
ExportLayerInfo targetInfo = layerTable [targetKey];

// change the color name and cut color number for this mapping
targetInfo.ColorName = "31";
targetInfo.CutColorNumber = 31;

// Set the change back to the map
layerTable [targetKey] = targetInfo;

// Set the modified table back to the options
dwgOptions.SetExportLayerTable (layerTable);

ICollection<ElementId> views = new List<ElementId>();
views.Add (view.Id);

exported = document.Export (Path.GetDirectoryName (document.PathName),
    Path.GetFileNameWithoutExtension (document.PathName), views, dwgOptions);
}
return exported;
}

```

### 5.16.2 导出 IFC (IFC Export)

Revit API 允许自定义应用程序重写 IFC 导出过程的默认实现。要创建自定义 IFC 导出程序，请实现 `IExporterIFC` 接口并用 `ExporterIFCRegistry` 类注册。

当 Revit 已注册 `IExporterIFC` 对象时，不管用户从 UI 还是从 API 方法 `Document.Export (String, String, IFCExportOptions)` 调用导出 IFC，都将使用它。两种情况下，如果没有已注册的自定义 IFC 导出程序，则使用 Revit 默认的导出 IFC 实现。

从 API 调用导出 IFC 时，可用 `IFCExportOptions` 设置导出选项，等同于用户从导出 IFC 对话框中设置。

#### 1. 注册 IFC 导出程序 (ExporterIFCRegistry)

代码 5-66 演示了使用 `OnStartup` 方法注册了一个自定义 IFC 导出程序。

#### 代码 5-66：注册自定义 IFC 导出程序

```

/// <summary>
/// This class implements the method of interface IExporterIFC to perform an export to IFC.
/// It also implements the methods of interface IExternalDBApplication to register the IFC export client to Autodesk Revit.
/// </summary>
class Exporter : IExporterIFC, IExternalDBApplication
{
    public ExternalDBApplicationResult OnStartup (Autodesk.Revit.ApplicationServices.ControlledApplication application)
    {

```



```

    ExporterIFCRegistry.RegisterIFCExporter (this);
    return ExternalDBApplicationResult.Succeeded;
}
}

```

注意: `ExporterIFCRegistry` 是一个应用程序级的单例模式, 且一个 Revit 给定会话只可注册一个 `IFCExporter`。注册是基于“服务第一”的原则。如果已经注册了一个 IFC 导出程序, 还调用 `RegisterIFCExporter`, 则会引发非法操作异常, 可以检查会话日志文件中是否有已成功注册的 IFC 导出程序的 DLL 路径。

## 2. IFC 导出程序接口 (IFCExporter)

接口 `IFCExporter` 只有一个方法来实现, 即 `ExportIFC()`。由 Revit 调用此方法以执行导出 IFC。`IFCExporter` 对象作为其参数之一传递给此方法。`IFCExporter` 是 Revit 提供的主类, 可以实现导出 IFC。它包含用户所选导出操作选项的信息, 以及用于存取正确执行导出所需的特定类型数据成员。

Autodesk.Revit.DB.IFC 命名空间, 包含大量 IFC 相关的 API 类, 可为 IFC 导出方法的自定义实现所用。自定义 IFC 导出应用的完整示例, 请参阅开放源代码示例: <http://sourceforge.net/projects/ifcexporter/>。

### 5.16.3 自定义导出 (Custom Export)

Revit API 提供了一组类, 可以通过自定义导出语境来导出三维视图。这些类提供对渲染输出渠道的访问, Revit 通过这些类将模型的三维图形表示发送到输出设备。在自定义导出情况下, “设备” 是一个语境对象, 可以表示任何类型的设备, 文件是最常见的“设备”。

自定义导出程序的实现提供了一个语境并调用模型渲染, 据此, Revit 开始处理模型, 并通过该语境方法发出图形数据。当渲染模型时, 这些数据将如实地以模型的形式在 Revit 中显现。这些数据包括所有几何图形和材料属性。

#### 1. 自定义导出程序类 (CustomExporter Class)

`CustomExporter` 类允许通过自定义导出语境来导出三维视图。该类的 `Export()` 方法触发 Revit 中的标准渲染过程, 只不过输出是通过给定的操作处理几何、非几何信息的自定义语境来引导, 而不是将结果展现在屏幕或打印机上。

#### 2. 导出语境接口 (IExportContext)

`IExportContext` 类的实例作为参数传入 `CustomExporter` 构造函数。然后, 调用此接口的方法作为模型实体导出。

#### 3. 渲染节点类 (RenderNode Classes)

`RenderNode` 是所有导出模型输出过程中的节点的基类。节点可以是几何图形如图元或灯光, 或非几何图形如材质。某些类型的节点为容器节点, 包括其他渲染节点。

#### 4. 照相信息 (CameraInfo)

`CameraInfo` 类描述三维视图渲染图像的投影映射信息。通过 `ViewNode` 属性可以得到此类的实例。如果为 “null”, 则应认为是正交视图。

## 附录 A 术语 (Glossary)

### 数组 (Array)

数组拥有一系列数据元素，这些元素通常具有相同的规模和数据类型。数组根据位置访问单个元素。位置由下标索引提供。索引通常以连续的整数或任何有序的值集来表示。

### 建筑信息模型 (BIM)

建筑信息模型是建设项目设计和建造中，运用和创建的相互协调、内部一致并适于计算的信息载体。建筑信息模型应用中，由信息衍生图形，但该信息不同于常规 CAD 中的原始信息。

### 类 (Class)

在面向对象的程序 (OOP) 中，类用于将相关的属性 (变量) 和方法 (函数) 一起进行分组。典型的类描述了这些方法怎样操作使用属性。类可以是独立的类或从其他类继承的类，对于后者，能派生其他类的类通常称为基类。

### 事件 (Events)

事件是应用程序内部发生一个事件时被调用的消息或过程。例如，存储或打开一个模型。

### 过滤器 (Iterator)

过滤器是一个对象，允许编程人员遍历一个集合 (数组、组集等) 中的所有元素，而不管其实现过程。

### 方法 (Method)

方法是类的函数或过程，用于操作或访问类的的数据成员，在编程中称之为函数。

### 命名空间 (Namespace)

命名空间是一个组织单元，用于把类似或功能相关的类组合到一起。

### 重载 (Overloading)

方法重载是指调用同名而具有不同类型或数量的传递参数的不同方法 (函数)。

### 属性 (Properties)

属性是类使用者可使用的类的的数据成员。在编程上称为变量。某些属性是只读的 [支持 `get()` 方法]，而另一些则是可修改的 [支持 `set()` 方法]。

### Revit 族 (Revit Families)

Revit 族是类型对象的集合。族将具有一组共同参数、相同用途和相似图形表示的图元进行分组。不同类型的族，其部分或全部参数可以有不同的值，但该组参数其名称和意义都相同。



### Revit 参数 (Revit Parameters)

Revit 参数有若干类型:

- 共享参数可以看作用户自定义变量。
- 系统参数是 Revit 中的硬编码变量。
- 族参数是在创建或修改族时所定义的变量。

### Revit 类型 (Revit Types)

类型是族的一个成员。对于模型中某类型的所有实例来说, 每个类型都有特定的常量参数, 称为类型属性。类型还有可以在模型中更改的参数, 称作实例参数。

### 集 (Sets)

集是一组没有特定顺序、不重复的值的合集 (容器)。除非有特殊的规约, 否则它与数学概念上的集合是相符的。

### 图元 ID (Element ID)

每个图元都对应一个 ID, 用一个整数值加以标识。在 AutoCAD Revit 项目中, 它提供了一种唯一地识别图元的方式。对于特定的项目它具有唯一性, 但跨不同 AutoCAD Revit 项目的图元 ID 不具有唯一性。

### 图元 UID (Element UID)

每个图元都对应有一个 UID, 它是一串全局唯一的标识符。这意味着即使跨不同 AutoCAD Revit 项目, 图元 UID 也是唯一的。

## 附录 B 疑问解答 (FAQ)

### B.1 常见问题 (General Questions)

问：如何参照 Revit 的图元？

答：每个图元都有一个 ID。在模型中 ID 具有唯一性，确保跨多个 Revit 会话也能引用同一图元。

问：一个模型只能使用一个共享参数文件吗？

答：共享参数文件用于保存有关参数的信息。最重要的信息是 GUID（全局唯一标识符），用于确保单个文件和跨多个模型中参数的唯一性。

Revit 可以使用多个共享参数文件，但是一次只能从一个文件中读取参数。可以选择所有模型使用同一共享参数文件，或是每个模型使用不同的共享参数文件。

此外，API 应用程序应避免影响用户参数文件。应用程序应附有其自身的参数文件来包含要用的参数。要将参数载入 Revit 文件，则：

- 应用程序必须知道用户参数文件名。
- 切换到应用程序参数文件并载入参数。
- 然后切换回用户文件。

问：为了其他程序可以使用共享参数，我需要随模型分发共享参数文件吗？

答：不需要。共享参数文件仅用于载入共享参数。在其载入之后，模型就不再需要该文件了。

问：共享参数值会随相应的图元一起拷贝吗？

答：是的。如果数据库中有一个共享参数，它拥有唯一图元 ID，请附加 Revit 图元唯一 ID 或向 Revit 图元唯一 ID 添加另一个共享参数。这样做，可以检查并确保正在使用的是原始图元 ID，而不是个拷贝。

问：图元唯一 ID (UID) 是全局唯一？可以改变吗？

答：图元 UID 是全局唯一的，但图元 ID 仅是模型内唯一。例如，如果将一面墙从某个 Revit 项目复制到另一个项目，墙的 UID 肯定改变，以保持全局唯一性，但墙的 ID 未必改变。

问：我的应用程序将数据发送回模型时，Revit 需要很长时间才能更新。我要怎么做才能加速？

答：确保只在必要时才调用 `Document.Regenerate()`。尽管需要使用这种方法来确保 Revit 文件中的图元反映所有更改，但会减慢应用程序的运行速度。还得记住，在提交事务时，会有一个重生成文件的自动调用。



问：如果我想将共享参数添加到某些图元，而这些图元没有绑定共享参数的可能，我该怎么办？例如，网格或材料。

答：如果某个图元类型不具备添加共享参数的可能，则需要添加一个项目参数。当需要访问该图元关联的共享参数时，这确实让它有点复杂，因为它并不显示为图元参数列表的一部分。通过使用技巧可能使参数与图元关联，如使项目中的共享参数为字符串，并让共享参数包括图元 ID，通过解析字符串将数据与图元关联。

问：我如何访问 BMP 格式保存的模型和内容？

答：Preview.dll 是一个 shell 插件，是实现 IExtractImage 接口的对象。IExtractImage 是一个接口，Windows Shell Folders 使用它来提取一个已知文件类型的图片。

有关更多信息，请回顾本书中的相关章节。

CRevitPreviewExtractor 实现标准 API 函数：

```
STDMETHOD (GetLocation) (LPWSTR pszPathBuffer,
                          DWORD cchMax,
                          DWORD *pdwPriority,
                          const SIZE *prgSize,
                          DWORD dwRecClrDepth,
                          DWORD *pdwFlags);
```

```
STDMETHOD (Extract) (HBITMAP*);
```

它在注册表中注册自己。

## B.2 Revit Structure 问题 (Revit Structure Questions)

问：有时候结构图元的默认端部约束释放致使模型不稳定。我怎么处理这种情况？

答：Revit Structure R3 中引入的分析模型检查功能可以找出其中一些问题。在导入分析模型期间，将提示是否要保留 RST (Revit Structure) 的约束释放条件，或是否要设置固定所有梁和柱。在将模型重新导入到 RST 时，每次都更新端部约束释放，不要覆盖后续导出到分析程序的端部约束释放。

问：我在转动梁的方向，于是它们被转向弱轴方向。例如，W14X30 “I” 梁旋转 90°，旋转后看起来像一根 “H” 梁。API 中的旋转角度是如何访问的？

由于定位是一条定位曲线而非一个定位点，我无法访问旋转值，那我需要检查什么？我有一个族实例图元要检查，要怎么做？

答：请看一下 SDK 中的 “RotateFramingObject” 示例。其中有如何获取和更改梁支撑和柱的旋转角度的例子。

问：我如何将新的混凝土梁和柱的尺寸添加到模型上？

答：请看一看 SDK 中的 “FrameBuilder” 示例代码。

问：我如何查看真实的压型板层？

答：SDK 中有一个名为 “DeckProperties” 的例子，提供了有关如何获取压型板的层信息。API 报告的压型板信息与 UI 中完全相同。压型板尺寸参数见图 B-1。



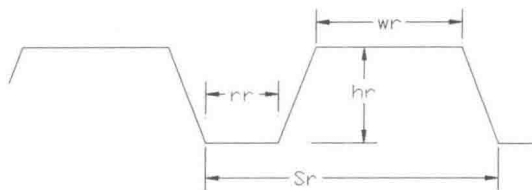


图 B-1 压型板尺寸参数

问：假如我有一根带有悬臂的梁，如何辨别它？

答：Revit 数据库中没有直接的方式辨别梁是否带有悬臂。然而，关于是否有部分是悬臂的，根据一个或多个以下选项，可以作出一个很好的判断。

有两个称作“Moment Connection Start”和“Moment Connection End”的参数。如果这两个参数的设置值不为“None”，那么应该看看是否有一根共线梁，且其设置值也不为“None”。同时，请用户务必选择“Cantilever Moment”选项而不是“Moment Frame”选项。

回溯检查该图元与其他图元的连接性。

查看图元约束释放条件。

问：当导出一个模型到外部程序，而模型中包含组，导出结束时用户收到以下错误信息：“只允许在组编辑模式更改‘组 1’。请使用**编辑组**命令来更改该组的所有实例。可以用‘Ungroup’选项，通过解组更改组实例，来继续这些更改”。

答：目前，API 不允许直接修改组成员，可以通过编程方式解组，进行更改并重新成组，然后将原组中的其他实例替换进新组，便可得到同样的效果。

## 附录 C VB.NET 代码的“Hello World”

本附录说明如何用 VB.NET 创建示例应用程序。该示例程序是使用 Microsoft Visual Studio 创建的。

### C.1 创建新项目 (Create a New Project)

使用 Visual Studio 编写 VB.NET 程序的第一步是选择一个项目类型并创建新项目。

- (1) 从“文件”菜单，选择**新建** ➤ **项目**....
- (2) 在已安装的“样板”框架中单击 **Visual Basic**。
- (3) 在右边框中，单击**类库**。应用程序假定项目位置是 D: \Sample。
- (4) 在“名称”字段，键入“HelloWorld ”作为项目名称，见图 C-1。
- (5) 单击 **OK**。

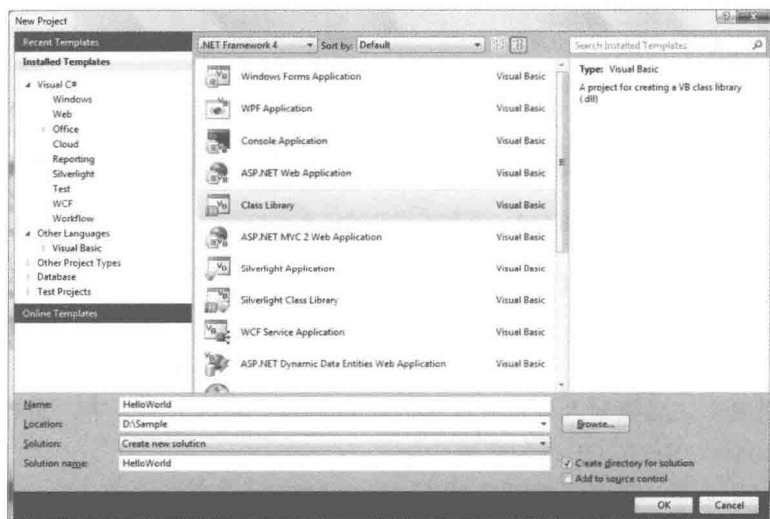


图 C-1 新项目对话框

### C.2 添加引用和命名空间 (Add Reference and Namespace)

VB.NET 使用过程类似于 C#。在创建“Hello World”项目之后，请完成以下步骤：

- (1) 在解决方案资源管理器中右键单击项目名称以显示上下文菜单。
- (2) 从上下文菜单中选择**属性**以打开属性对话框。在属性对话框中，单击**引用**选项卡。



会出现一个引用和命名空间的列表。

(3) 单击**添加**按钮以打开添加引用对话框。

(4) 在添加引用对话框中，单击**浏览**选项卡。找到 **Revit** 安装文件夹，然后单击 **RevitAPI.dll**。例如，安装文件夹位置可能是 **C:\Program Files\Autodesk\Revit Architecture 2012\Program\RevitAPI.dll**，见图 C-2。

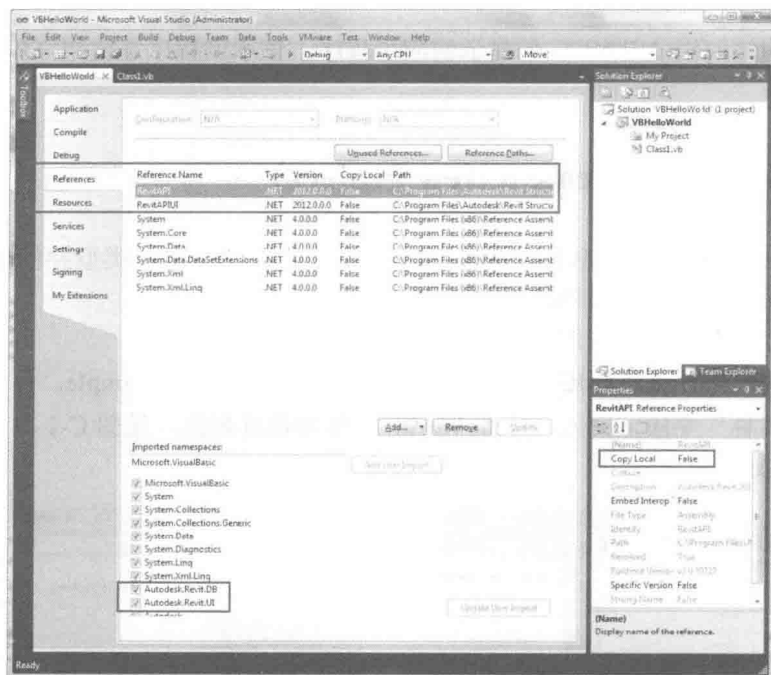


图 C-2 添加引用及导入命名空间

(5) 单击 **OK** 以添加引用并关闭对话框。

(6) 重复上述步骤以添加一个 **RevitAPIUI.dll** 引用，它与 **RevitAPI.dll** 在同一文件夹中。

(7) 添加引用后，必须导入项目中使用的命名空间。在这个例子中，是导入 **Autodesk.Revit.DB** 和 **Autodesk.Revit.UI** 命名空间。

(8) 要完成该过程，在引用框架中单击 **RevitAPI** 以高亮显示它。在属性框设置副本为“false”。对 **RevitAPIUI.dll** 重复这一步骤。

### C.3 更改类名 (Change the Class Name)

更改类名，请完成以下步骤，见图 C-3：

(1) 在解决方案资源管理器中，右键单击 **CLASS1.vb** 以显示上下文菜单。

(2) 从上下文菜单中，选择**重命名**。将文件重命名为“**HelloWorld.vb**”。

在解决方案资源管理器中，双击 **HelloWorld.vb** 打开它进行编辑。

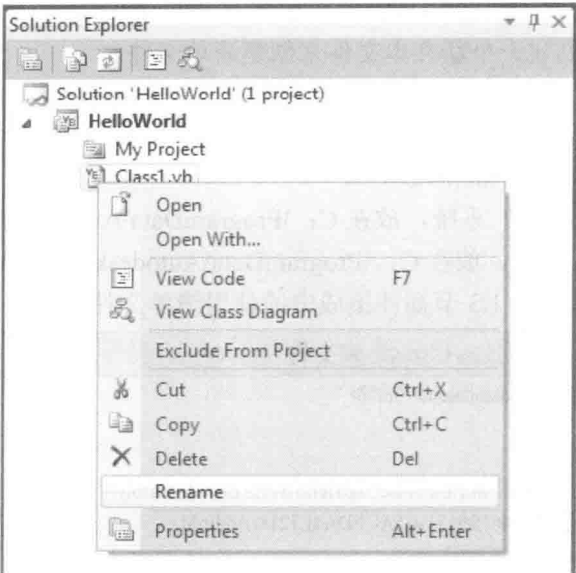


图 C-3 更改类名

### C.4 添加代码 (Add Code)

在 VB.NET 中编写代码时，必须注意到关键字用大写，见代码 C-1。

代码 C-1: Hello World 的 VB.NET 代码

```
Imports System

Imports Autodesk.Revit.UI
Imports Autodesk.Revit.DB

<Autodesk.Revit.Attributes.Transaction (Autodesk.Revit.Attributes.TransactionMode.Automatic) > _
Public Class Class1
Implements IExternalCommand

    Public Function Execute (ByVal revit As ExternalCommandData, ByRef message As String,
        ByVal elements As ElementSet) As Autodesk.Revit.UI.Result _
        Implements IExternalCommand.Execute

        TaskDialog.Show ("Revit", "Hello World")
        Return Autodesk.Revit.UI.Result.Succeeded

    End Function
End Class
```

### C.5 创建.addin 清单文件 (Create a .addin Manifest File)

项目输出目录中会出现 HelloWorld.dll 文件。如果希望在 Revit 调用应用程序，则必须



创建清单文件来注册到 Revit。

1. 可在记事本中创建一个新文本文件来创建清单文件。
  2. 添加代码 C-2 到文本中，将文件保存为 HelloWorld.addin，并放在以下位置：
    - 对于 Windows XP 系统，放在 C:\Documents and Settings\All Users\Application Data\Autodesk\Revit\Addins\2012\。
    - 对于 Vista/Windows 7 系统，放在 C:\ProgramData\Autodesk\Revit\Addins\2012\。
    - 对于 Windows8 系统，放在 C:\ProgramData\Autodesk\Revit\Addins\2012\。
- 有关详细信息，请参阅 1.3 节插件集成中的使用清单文件。

#### 代码 C-2：为外部命令创建 .addin 清单文件

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<RevitAddIns>
  <AddIn Type="Command">
    <Assembly>D:\Sample\HelloWorld\bin\Debug\HelloWorld.dll</Assembly>
    <AddInId>239BD853-36E4-461f-9171-C5ACEDA4E721</AddInId>
    <FullClassName>Class1</FullClassName>
    <Text>HelloWorld</Text>
    <VendorId>ADSK</VendorId>
    <VendorDescription>Autodesk, www.autodesk.com</VendorDescription>
  </AddIn>
</RevitAddIns>
```

## C.6 生成程序 (Build the Program)

完成了代码之后，必须生成程序文件。从“构建”菜单，单击**生成解决方案**。生成输出将显示在输出窗口中，其指示项目编译没有错误。

## C.7 调试程序 (Debug the Program)

在调试模式下运行程序，并使用断点暂停程序，以便检查变量和对象的状态。如果出现错误，检查程序运行的变量，推断其值为什么与预期的不同。

- (1) 在解决方案资源管理器窗口中，右键单击 **HelloWorld** 项目以显示上下文菜单。
- (2) 从上下文菜单中，单击**属性**。此时将显示属性窗口。
- (3) 单击**调试**选项卡。
- (4) 在调试窗口中启动操作部分，单击**启动外部程序**浏览 Revit.exe 文件，见图 C-4。默认情况下，该文件位于 C:\Program Files\Autodesk\Revit Structure 2012\Program\Revit.exe。
- (5) 从调试菜单中，选择触发断点（或按 F9 键）在代码 C-3 上设置断点。

#### 代码 C-3：任务对话

```
TaskDialog.Show ("Revit", "Hello World")
```

- (6) 按 F5 键启动调试程序。



(7) 测试调试。

- 在插件选项卡上会出现“HelloWorld” 外部工具菜单按钮，见图 C-5。
- 单击 **HelloWorld** 来执行程序，激活断点。
- 按 F5 键继续执行程序，将显示图 C-6 所示系统消息。

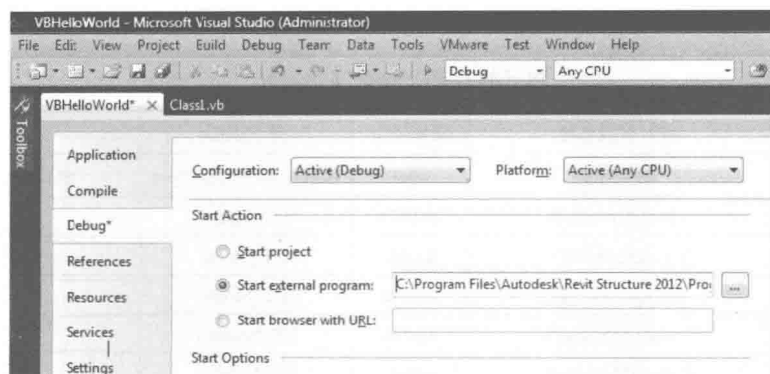


图 C-4 设置调试环境

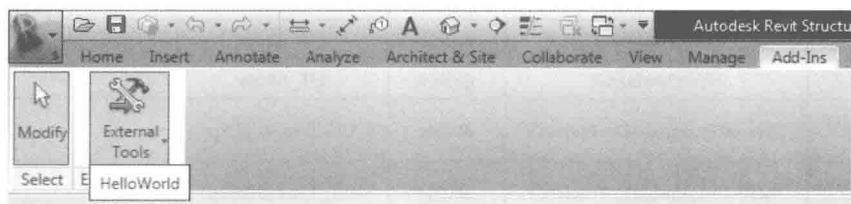


图 C-5 HelloWorld 外部工具命令

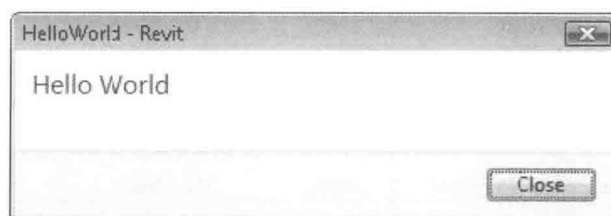


图 C-6 任务对话消息

## 附录 D 内部单元的材料属性 (Material Properties Internal Units)

表 D-1 单元材料基本参数 (钢材)

参数名称	API 参数名称	变量类型	单位类型	说明
Behavior	Behavior	bool	Isotropic, Orthotropic	目前通用参数可用
Young Modulus X	YoungModulusX	double	UT_Stress	
Young Modulus Y	YoungModulusY	double	UT_Stress	
Young Modulus Z	YoungModulusZ	double	UT_Stress	
Poisson ratio X	PoissonModulusX	double	UT_Number	
Poisson ratio Y	PoissonModulusY	double	UT_Number	
Poisson ratio Z	PoissonModulusZ	double	UT_Number	
Shear modulus X	ShearModulusX	double	UT_Stress	
Shear modulus Y	ShearModulusY	double	UT_Stress	
Shear modulus Z	ShearModulusZ	double	UT_Stress	
Thermal Expansion coefficient X	ThermalExpansionCoefficientX	double	UT_TemperaExp	
Thermal Expansion coefficient Y	ThermalExpansionCoefficientY	double	UT_TemperaExp	
Thermal Expansion coefficient Z	ThermalExpansionCoefficientZ	double	UT_TemperaExp	
Unit Weight	UnitWeight	double	UT_UnitWeight	
Damping ratio	DampingRatio	double	UT_Number	
Minimum Yield Stress	MinimumYieldStress	double	UT_Stress	US 代号 Fy, 亦可认为是压应力能力
Minimum tensile stress	MinimumTensileStrength	double	UT_Stress	仅用于钢材
Reduction factor for shear	ReductionFactor	double	UT_Number	剪切屈服极限应力折减系数。 剪切屈服极限应力 = MinimumYieldStress / ReductionFactor

表 D-2 单元材料基本参数 (混凝土)

参数名称	API 参数名称	变量类型	单位类型	说明
Behavior	Behavior	bool	Isotropic, Orthotropic	目前通用参数可用
Young Modulus X	YoungModulusX	double	UT_Stress	
Young Modulus Y	YoungModulusY	double	UT_Stress	
Young Modulus Z	YoungModulusZ	double	UT_Stress	
Poisson ratio X	PoissonModulusX	double	UT_Number	



续表

参数名称	API 参数名称	变量类型	单位类型	说明
Poisson ratio Y	PoissonModulusY	double	UT_Number	
Poisson ratio Z	PoissonModulusZ	double	UT_Number	
Shear modulus X	ShearModulusX	double	UT_Stress	
Shear modulus Y	ShearModulusY	double	UT_Stress	
Shear modulus Z	ShearModulusZ	double	UT_Stress	
Thermal Expansion coefficient X	ThermalExpansionCoefficientX	double	UT_TemperalExp	
Thermal Expansion coefficient Y	ThermalExpansionCoefficientY	double	UT_TemperalExp	
Thermal Expansion coefficient Z	ThermalExpansionCoefficientZ	double	UT_TemperalExp	
Unit Weight	UnitWeight	double	UT_UnitWeight	
Damping ratio	DampingRatio	double	UT_Number	
Concrete compression	ConcreteCompression	double	UT_Stress	混凝土压应力能力。 美国标准编码 F'c
Lightweight	LightWeight	Bool		若为“true”则定义为 轻质混凝土
Shear strength modification	ShearStrengthReduction	double	UT_Number	若 Lightweight = true, 则此值 可用。它是混凝土压剪能力的 折减。混凝土抗剪应力= ConcreteCompression / ShearStrengthReduction

表 D-3 单元材料基本参数 (木材)

参数名称	API 参数名称	变量类型	单位类型	说明
Young Modulus	PoissonModulus	double	UT_Stress	
Poisson ratio	ShearModulus	double	UT_Number	
Shear modulus	ShearModulus	double	UT_Stress	
Thermal Expansion coefficient	ThermalExpansionCoefficient	double	UT_TemperalExp	
Unit Weight	UnitWeight	double	UT_UnitWeight	
Species	Species	AString		
Grade	Grade	AString		
Bending	Bending	double	UT_Stress	
Compression parallel to grain	CompressionParallel	double	UT_Stress	
Compression perpendicular to grain	CompressionPerpendicular	double	UT_Stress	
Shear parallel to grain	ShearParallel	double	UT_Stress	
Tension perpendicular to grain	ShearPerpendicular	double	UT_Stress	





表 D-4

单位类型表达

序号	单位类型	量纲	内部表示方法
1	UT_Number	无	单值
2	UT_Stress	$(\text{质量}) \times (\text{长度}^{-1}) \times (\text{时间}^{-2})$	$\text{kg}/(\text{ft} \cdot \text{s}^2)$
3	UT_TemperalExp	$(\text{温度}^{-1})$	$1/^\circ\text{C}$
4	UT_UnitWeight	$(\text{质量}) \times (\text{长度}^{-2}) \times (\text{时间}^{-2})$	$\text{kg}/(\text{ft} \cdot \text{s}^2)$

## 附录 E API 用户界面指南

### (API User Interface Guidelines)

#### E.1 引言 (Introduction)

本附录旨在对基于 Revit API 的应用程序提供设计用户界面方面的指导和最佳做法。欧特克设计和开发人员在开发 Revit 时，遵循以下原则，并为掌握本书其余部分打下良好基础。

#### E.2 一致性 (Consistency)

对基于 API 的应用程序，重要的是要让用户体验到与 Revit 的一致性。这将让应用程序用户可以利用它们学习 Revit 期间所获得的知识。通过再利用某些类型的对话框可以获得一致性，来替代创建或重新创建对话框。请参见 E.8.5 节对话框类型，其中的对话框示例，在创建应用程序时可以利用。

#### E.3 使用用户语言 (Speak the Users' Language)

以用户的语言理解和交流是非常重要的，特别是在用户界面内。用户需要提供和接收来自应用程序的信息反馈。

用户界面语言中使用的语气应该是通俗、帮助、协商的语气。用户界面应该亲和并向用户提供清晰、翔实的信息，便于用户准确操作。

#### E.4 高质量布局 (Good Layout)

通过以下格式塔原则，可以获得信息的均衡布局。

- 接近性：靠在一起的项目，被认为是密切相关的。
- 相似性：有着相似外观的项目，被认为是密切相关的。
- 连续性：相比更加复杂，但似乎也是可行的形式，人们往往更喜欢简单、完整的外形。

#### E.5 设好默认值 (Good Defaults)

当用户需要编辑数据或更改设置时，在缺乏任何或明显默认值的情况下，会导致出错并强制用户重新编辑和重新输入信息。请记住：



- 可根据咨询的惯用数据或利用先前由用户输入的值, 以一个合理的默认值为当前上下文设置控件的值。适当的默认值也可能为空值。
- 在适当情况下, 要记住用户上次使用的设置, 而非总是使用系统提供的同一默认值。

一个常见的例子是, 为最常选用的某些设置或选项, 进行预先设置。

## E.6 渐进式展开 (Progressive Disclosure)

随着应用程序复杂性的增加, 用户就越难找到他们所需要的。为了适应用户界面的复杂性, 通常将数据或选项分成组群。以渐进展开的概念, 在必要时显示所需的信息。渐进展开可能由用户引发、系统引发, 或两者混合引发。

### E.6.1 用户引发操作 (User Initiated Action)

这个例子包括用于启动子对话框的一个**显示更多**按钮, 将组块界面元素放入逻辑块的一些选项卡, 以及按一定条件选择性地显示集合中的项的集合搜索/过滤。

### E.6.2 系统引发操作 (System Initiated Action)

这些可以是:

- 基于事件: 产品内部的事件引发更多信息披露, 如使用任务对话框。
- 基于时间: 在规定的時間之后引发更多的信息披露, 如自动幻灯片放映。

### E.6.3 混合引发 (用户和时间) [Hybrid (User and Time Initiated)]

渐进式提示是混合引发的典型例子, 用户通过将鼠标悬停于控件上引发初始提示, 但在一段时间之后, 会出现更详细的提示。

## E.7 本地化用户界面 (Localization of the User Interface)

如果本地化用户界面计划使用英语以外的语言, 请注意空间要求。

英文语言非常紧凑, 所以翻译文本最终会占用更多空间 (对长串字符平均来说 30%, 对单词或短语这样的短字符串, 100%或更多)。如果翻译文本插入为英文产品设计的对话框中, 一般都会出现问题, 通常是因为没有足够的可用空间适应翻译的文本。通常解决这一问题的办法是调整对话框尺寸, 以使翻译好的文本正好合适, 但大多数时候这并非最佳解决方案。

相反, 通过开发人员精心设计的对话框, 可以对大部分语言使用相同的对话框资源, 而不需要昂贵和耗时的重新设计。本书描述如何设计“全球”对话框。

必须时刻遵守以下设计规则, 以防止全球化和本地化的问题。

- 英语对话必须至少小于产品指定的最小屏幕尺寸的 30%。
- 对话框设计时必须考虑到以下数量的扩展空间 (表 E-1)。这个额外的空间应该符合英语的界面显示要求, 且能避免绝大部分界面语言本地化过程中的空间调整。



表 E-1 对话框扩展空间

字符	百分比/%	字符	百分比/%
1~5 字符	100	11~100 字符	30
6~10 字符	40	100 字符以上	20

- 使文本控件周围的无形控制框架、图文框等尽可能地大，以允许长翻译文本。这些框架应至少大于英文文本的 30%。

请参阅“对微软 Windows 用户体验的指导方针之用户界面文本指引”，以获得更多有关本地化文本的信息。

## E.8 对话框指南 (Dialog Guidelines)

### E.8.1 引言 (Introduction)

对话框是应用程序的一个单独的可控区域，包含用于信息编辑的控件。对话框是获取用户输入和显示信息的主要载体。在决定使用哪个对话框时，请参照表 E-2。

表 E-2 对话框类型选用条件

对话框类型	定 义	何时使用
模态	暂停应用程序的其余部分，并等待用户输入	对话框中的任务不常发生
非模态	用户可在对话框和应用程序的其余部分之间切换，而不用关闭对话框	对话框中的任务频繁发生，拦截系统会中断用户 workflow

### E.8.2 行为准则 (Behavior Rules)

- 对话框可以由用户引发（通过单击控件提示），也可以由系统引发（由系统触发事件警告）。
- 初始对话位置通常为屏幕中心，但是这个规则可能因使用变化或使用特定的上下文而改变，但应该有一个即时焦点。
- 当对话框内容动态变化时，对话框应可调整大小，关于这一主题的更多内容，请参阅 E.8.4.7 节动态布局。

### E.8.3 对话框控件 (Dialog Controls)

对话框控件使用示例见表 E-3。

表 E-3 对话框控件使用示例

控件	何时使用	示 例
复选框	可以通过单个标签向用户清楚地表达做出的选择（及其反选）	<input checked="" type="checkbox"/> Enable Recent Files page at startup 启用的反选为禁用
单选按钮	有 2~5 项互斥却又相关的选择，而用户每次只能做出一个选择	New pages should be opened in: <input type="radio"/> a new window <input checked="" type="radio"/> a new tab



续表

控件	何时使用	示 例
文本框	这种选择需要手动输入一个数值文本值	
下拉列表	从互斥选项列表中选择, 适用于隐藏其余选择, 而只显示默认选择。 此外, 在有超过四个选项或实际空间受限时, 使用下拉列表而不是单选按钮	
组合框	类似下拉列表框, 但允许用户输入非预先填入下拉列表的信息	
滑块	当期望的输入或现有数据在特定范围内时使用。 还可以将滑块与文本框相结合, 给用户更高层级的控件并在移动滑块时给出信息反馈	
微调按钮	如果数据可以循序输入和逻辑限制, 则使用此选项。可以连同一个可编辑的文本框一起使用	

## E.8.4 铺设对话框 (Laying Out a Dialog)

### E.8.4.1 基本要素 (Basic Elements)

每个对话框都包含的基本要素见图 E-1。

#### 1. 标题栏

标题栏文本描述窗口内容。

#### 2. 帮助按钮

帮助按钮是标题栏中关闭按钮旁边的一个按钮。

- 帮助按钮是可选的, 仅在有关联文档说明时使用。
- 请注意, Revit 中许多传统对话框还在对话框右下角有帮助按钮。

#### 3. 控件

大部分对话框包含用于更改设置或与相关应用程序内部数据交互的控件。控件的布局应遵循布局流程、间距和边距、分组及对话框控件等以下各节中所述规则。

当控件与另一个控件交互操作时, 如文本框和一个浏览按钮, 通过将相关联的控件放



置在以下 3 个位置之一，来标志控件之间的关系：



图 E-1 基本要素

- 在右上方与其他控件对齐。
- 在左下方与其他控件对齐，参见 E.8.5.4 节内容编辑器。
- 相关控件垂直中心线，参见 E.8.13.5 节列表生成器。

#### 4. 提交按钮

请参见 E.8.14.4 节提交更改。最常用的提交按钮包括：

- OK。
- Cancel。
- Yes。
- No。
- Retry。



- Close。

#### E.8.4.2 布局流程 (Layout Flow)

用户在用户界面查看对话框中时，有多项任务要完成。信息的设计和布局必须支持用户完成他们的任务。记住这一点，重要的是要考虑用户体验：

- 浏览（非读取）界面，当他们已获得想要的就中止，而忽略其他任何事情。
- 关注不同的项目。
- 尽量不要滚屏。

在任务的“开始”“中间”“结束”工作流中，建议以“线性路径”的方式布局窗口。这个路径应该设计得简明扼要。“中间”工作流的“线性路径”应能清楚地说明各控件之间的逻辑关系。当然，在完成任务时，并非所有用户都遵循一个严格的线性路径。这个路径适用于特定的工作流。

1. 变化 A：由上而下布置 UI 项 (图 E-2)

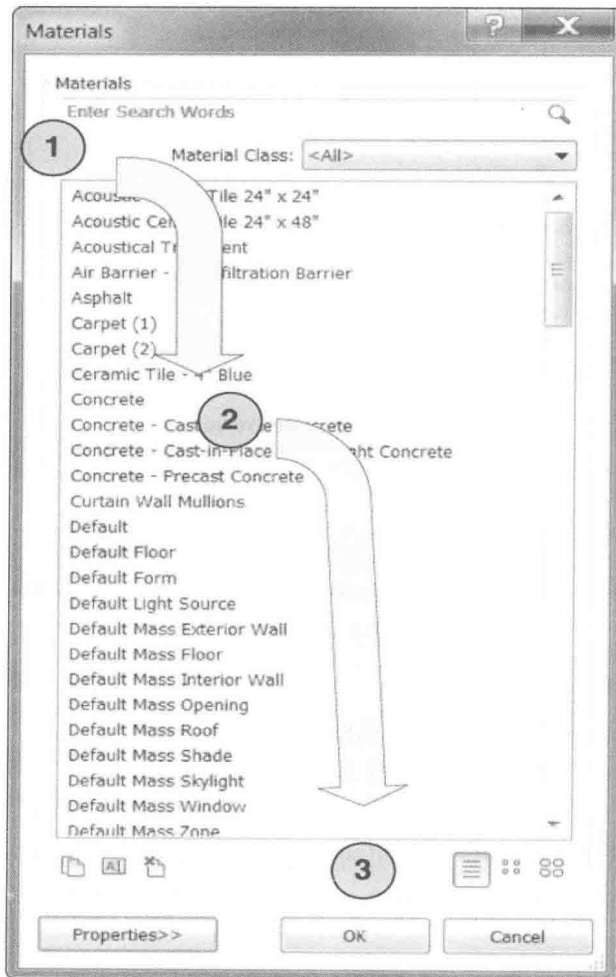


图 E-2 材料对话框

- (1) 启动左上角或顶部中央的任务。



(2) 用户须在中央区域与程序交互以完成任务。

(3) 在左下角结束任务。

本例中(材料对话框)用户执行以下操作:

- 搜索/过滤列表。
- 选择一个项。
- 提交或取消。

2. 变化 B: 自左至右布置 UI 项 (图 E-3)

(1) 启动任务或在最左边选项之间更改挑选 (参阅附录 E.8.14.1 节选项卡)。

(2) 中部是个独立分区, 用户须在该区域与程序交互以完成任务。

(3) 在右下角结束任务。

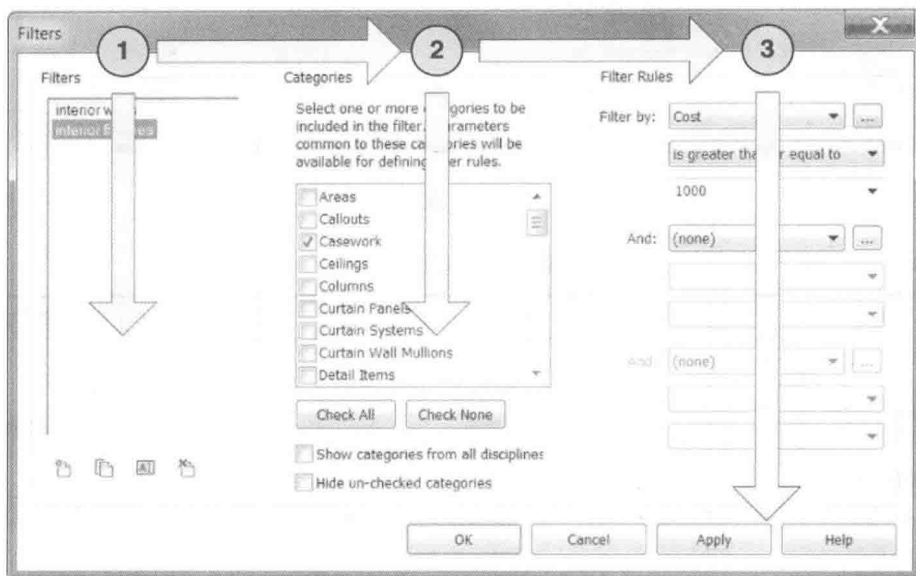


图 E-3 打开 Revit 文件对话框

注: 如果用户浏览的不是左边的快捷方式而是文件的层次结构, 则该对话框内也可以使用由上而下的流程。

3. 变化 C: 混合布置 (图 E-4)

像在变化 B 中所看到的那样, 许多自左至右布局的窗口, 其实是一种自左至右和由上而下的混合布局。

在这个 Revit 过滤器例子中, 三个主要任务按栏分组, 通过分组框圈定为: 过滤、类别和过滤规则。每栏包含一个由上而下的流程, 其中包括项目集合中的选择或修改控件。

#### E.8.4.3 间距和边距 (Spacing and Margins)

表 E-4 列出了由微软 Windows 用户体验指南之布局部分列表推荐的常见 UI 元素之间的间距 (用户界面为分辨率 96dpi, 字体为微软雅黑 9 磅)。对话框单位 (DLU) 与相应像素之间的差异定义, 请参阅微软 Windows 用户体验指南中的演练: 设计有空白边距、能自动缩放的 Windows 窗体控件。



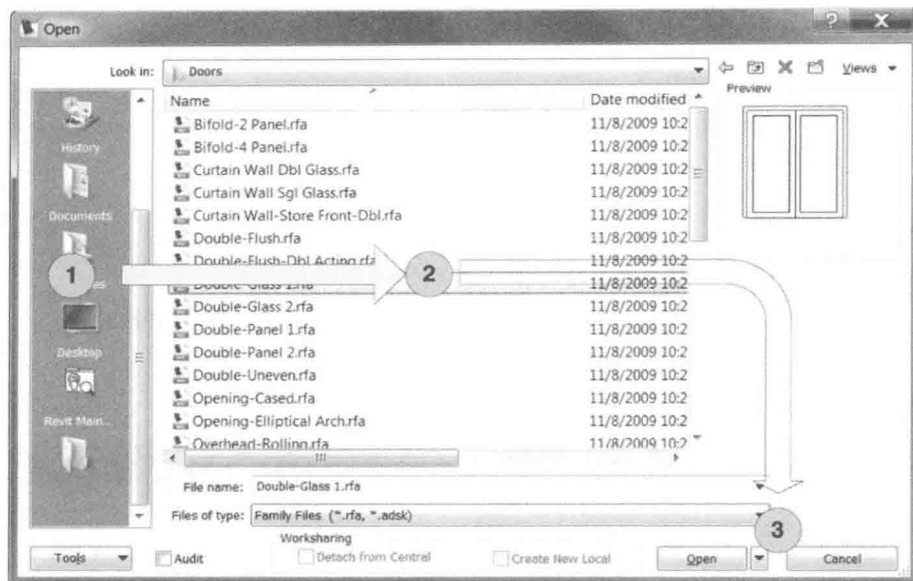


图 E-4 Revit 过滤器对话框

表 E-4

间距和边距设置表

要素	布置	对话框单位	相应像素
	对话框边距	7 单位每边	11 像素每边
	在文本标签和相关控件之间 (如文本框和列表框)	3	5
	相关控件之间	4	7
	不相关的控件之间	7	11
	分组框的第一个控件	分组框顶部向下 11; 与框的标题垂直对 齐	分组框顶部向下 16; 与框的标题垂直对齐
	分组框控件之间	4	7



续表

要素	布置	对话框单位	相应像素
	水平或垂直排列的按钮之间	4	7
	分组框最后一个控件	分组框底部向上 7	分组框底部向上 11
	自分组框左边缘	6	9
	控件旁边的文本标签	控件顶部向下 3	控件顶部向下 5
	文本段落之间	7	11
	交互式控件之间最小间距	3 或无间距	5 或无间距
	非相关控件和任何其他控件之间最小间距	2	3
	当控件依赖于另一个控件时, 它应当缩进 12DLU 或 18 像素, 这是复选框和单选按钮标签之间的设计距离	12	18

#### E.8.4.4 分组 (Grouping)

分组框是最常见的解决方案, 用于将对话框中的有关联性的控件成组, 见图 E-5。

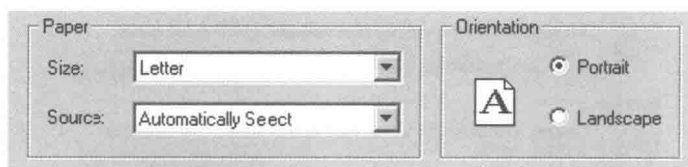


图 E-5 标准打印对话框内的分组框

分组框应包括:

- 两个或更多相关控件。
- 至少还有一个其他分组框。
- 标签:
  - 描述分组。
  - 遵循句型:



- ◆ 用名词或名词短语形式。
- ◆ 不使用结束标点符号。
- 符合间距规则的间距和边距部分。

#### E.8.4.5 设计欠佳的示例——避免使用 (Poor Examples– What Not to Use)

以下是设计欠佳的例子：

- 分组框没有标签或分组框只有一个控件，见图 E-6。



图 E-6 只有一个控件的分组框

- 一个对话框中只有一个分组框。
- 有标题的对话框，其单一分组框标题（材料）是多余的，可以删除掉。

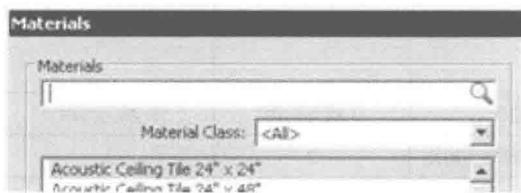


图 E-7 分组框标题多余

如图 E-7 所示，分组框没有标签且只有一个控件，而且分组框的标题相对于对话框标题是多余的。

要避免多个分组框两两之间互相“嵌套”，并避免把提交按钮放入分组框。

#### E.8.4.6 水平分隔符 (Horizontal Separator)

水平分隔符可以替代传统的分组框。仅当分组是垂直叠放在单个列中时可用。

以下示例取自 Microsoft Outlook 2007，见图 E-8。

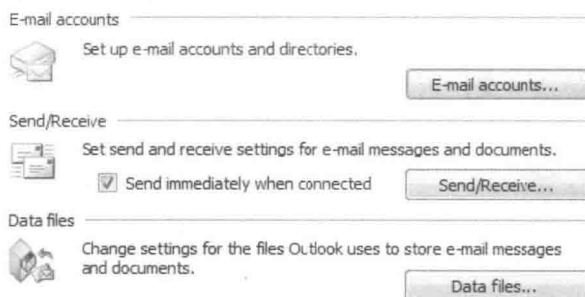


图 E-8 Microsoft Outlook 2007 中的水平分隔符

前一分组最后一个控件和后一分组线之间的间距应为 12DLU (18 个相对像素)。

#### E.8.4.7 动态布局 (Dynamic Layout)

不同类型或大小的显示设备所呈现的内容，通常需要能够适应显示窗体。使用动态布局有助于适应不同使用环境，如本地化为其他语言、改变内容的字体大小，并能让用户手动扩展窗口以查看更多信息。



创建一个动态布局，需要：

- 除非受限制，否则窗体内容应动态适应容器大小的改变。
- 在该对话框的右下角添加一个调整框体大小的手柄。
- 对话框尺寸不应小于默认值。
- 窗口大小应该符合用户在应用程序语言中定义的值。
- 在对话框调整大小时，其中的元素应根据表 E-5 中描述的象限保持对齐。

表 E-5

象 限 调 整

原位象限	对齐
一	左上
二	右上
三	左下
四	右下
Multiple	若控件位于多象限中，则应固定于象限一和/或象限三（左侧），并扩大/缩小到右侧以保持对齐

在这个示例中，列表框位于所有 4 个象限。因此，它是固定在左上方，并扩展到右下方，见图 E-9。

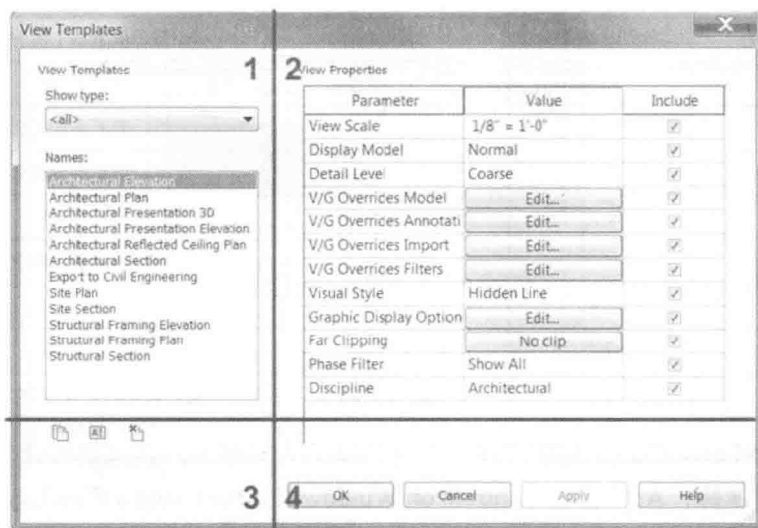


图 E-9 四方形网格应用于 Revit 视图样板对话框以说明应如何调整大小

#### E.8.4.8 实现说明 (Implementation Notes)

下面是实现动态布局时应考虑的一些步骤：

- 当改变容器的大小时，应先中断基于结构内容部分的设计和开发人员想要完成的流程。
- 应定义各区域和所用控件的最小、最大和其他尺寸限制。这通常取决于我们提供的数据类型、图像、补充信息和控件的目的。
- 应考虑对齐方式，也就是说，当重新调整大小时，其内容将如何流动。还应考虑哪



些项目是静态的，哪些是动态的，以及它们将如何扩展——对于从左向右的语言文字，对话框的流向通常也应该从左向右、自上而下，对齐并固定到左上方。对于不同的控件，调整大小的控制准则，请参照表 E-6。

表 E-6 控 件 属 性

控件	内容	可调整大小	可移动
按钮	静态	否	是
链接	静态	否	是
单选按钮	静态	否	是
微调控件	静态	否	是，基于附属元素
滑块	静态	X 方向	是
滚动条	静态	X 方向	是
选项卡	动态	X、Y 方向	是，但不小于所含最大控件
渐进式展开	动态	X、Y 方向	是，但不小于所含最大控件
复选框	静态	否	是
下拉列表	动态	X 方向	是，但不小于所含最大文本
组合框	动态	X、Y 方向	是，但不小于所含最大文本
列表浏览	动态	X、Y 方向	是，但不小于所含最大文本
文本框	动态	X、Y（多行）	是
日期时间框	动态	X 方向	是，但不小于所含最大文本
树状图	动态	X、Y 方向	是，但不小于所含最大文本
画布	动态	X、Y 方向	是
分组框	动态	X、Y 方向	是，但不小于所含最大控件
进度条	静态	X 方向	是
状态条	动态	X 方向	是
表或数据网格	动态	X、Y 方向	是，表列应在 X 方向上按比例增长

关于用 FlowLayoutPanel 创建可调大小对话框的更多细节，建议参阅微软 Windows 用户体验指南中的演练：Arranging Controls on Windows Forms Using a FlowLayoutPanel。

E.8.5 对话框类型 (Dialog Types)

Revit 默认设有一些对话框类型。利用这些标准类型有助于对话框的一致性，并可利用用户已学到的知识。

E.8.5.1 标准输入对话框 (Standard Input Dialog)

这是最基本的对话框类型。在用户需要做出某些选择，然后基于这些选择执行一个独立操作的情况下，应该使用该对话框。控件应遵循分组、间距和边距以及布局流程规则。

Revit 导出二维 DWF 格式文件选项对话框是一个比较好的标准输入对话例子，见图 E-10。

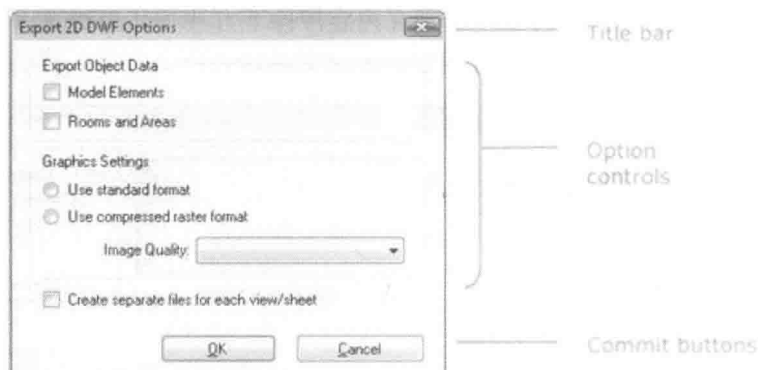


图 E-10 导出二维 DWF 选项对话框

### E.8.5.2 属性编辑器 (Property Editor)

在用户需要修改项目属性时使用。要创建属性编辑器，请提供一个显示名称/属性的表视图，见图 E-11。属性字段可以用文本框、复选框、命令按钮、下拉列表，甚至滑块来做修改。

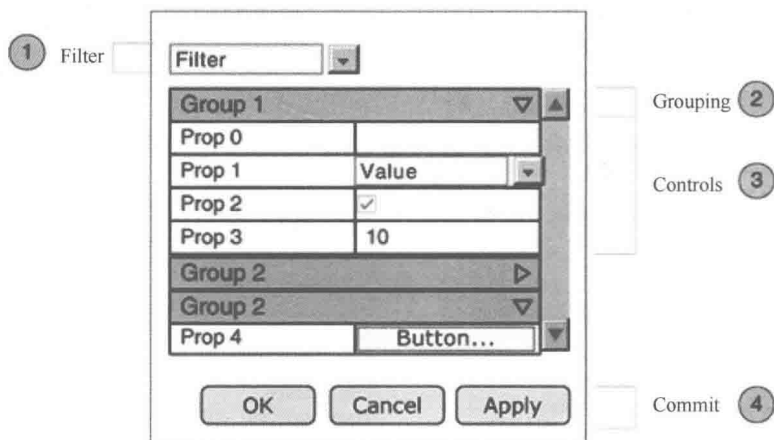


图 E-11 属性表格

### E.8.5.3 受支持的操作 (Supported Behaviors)

属性编辑器受支持的操作一览表见表 E-7。

表 E-7 属性编辑器受支持的操作一览表

编号	操作	说 明	是否必需
1	过滤	根据给定条件过滤属性列表	否
2	分组	对属性分组，以便易于浏览	否
3	控件 (编辑某项属性)	每个值单元格可以包含一个可编辑 (或禁用) 控件，这取决于上下文	是
4	提交 (按钮)	可选项，仅用于模态对话框	否

### E.8.5.4 内容编辑器 (Content Editor)

如果多个操作控件与同一内容控件 (如列表框) 交互，可与内容控件右上角对齐并垂



直堆叠这些控件,也可将这些控件水平布置在内容控件下方并与其左对齐,见图 E-12。具体采用哪种布局方式可由开发人员自行决定。

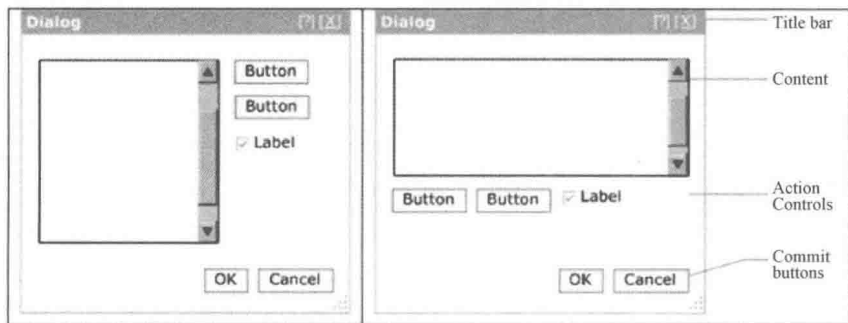


图 E-12 内容控件外右方和下方的操作控件

### E.8.5.5 集合浏览器 (Collection Viewer)

在一些如 Revit 的应用程序中,用户必须查看和管理项目集合来完成任务。集合视图为用户提供了多种在集合中查看项目的方式(浏览、搜索和过滤)。要提供一种便于用户浏览项目集合的方式,以选择某个项目来查看或编辑(参见 E.8.13.2 节集合编辑器),并有选择地提供集合搜索或过滤功能。

图 E-13 展示了一个列表集合浏览器。表视图和树状视图也是显示集合项的选项。

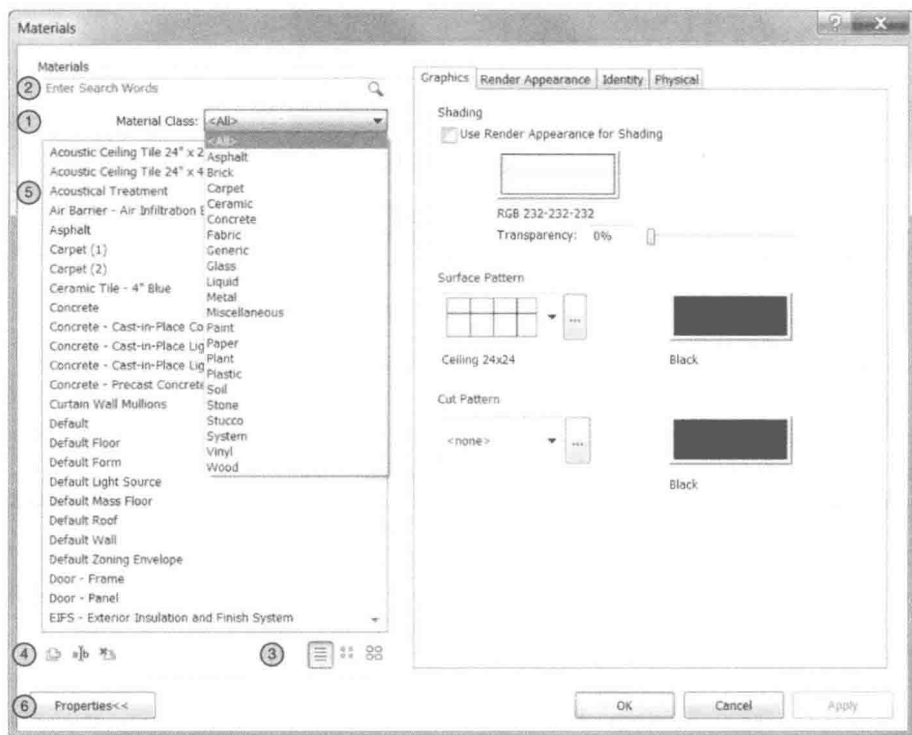


图 E-13 Revit 材料对话框



### E.8.5.6 受支持的操作 (Supported Behaviors)

集合浏览器受支持的操作一览见表 E-8。

表 E-8 集合浏览器受支持的操作一览表

编号	操作	说 明	是否必需
1	过滤	提供过滤该列表的选项，可表示为一个： <ul style="list-style-type: none"> <li>下拉——必须选择默认条件且包括一个“全部”选项。</li> <li>复选框——复选框“ON”将细化列表。默认由设计者设置。</li> <li>单选按钮——在单选按钮之间选择细化列表。默认由设计者设置一次选择一个，集合将根据所选项自动更新。控件必须遵循微软 Windows 用户体验指南</li> </ul>	否
2	搜索框	搜索框允许用户在集合中搜索关键字。搜索框必须遵循微软 Windows 用户体验指南	否
3	更改查看模式	如果集合被视为列表，可以选择将文本项目替代为大图标或小图标显示	否
4	集合管理	单独的 UI 将用于向集合中编辑、重命名、删除或添加项目。请参阅 E.8.13.2 节集合编辑器。它仅用于管理用户可编辑的集合	否
5	查看集合	可用下列方式查看集合本身：列表视图、表格视图、树状图或树状表格	是
6	显示更多内容	此按钮隐藏/显示与当前所选项关联的附加数据。请参见 E.8.14.3 节中的显示更多按钮内容	否

### E.8.5.7 列表视图 (List View)

当用户需要查看、浏览并随意选择、排序、分组、过滤或搜索项目的单一集合时，如果列表是分层的，则使用树状图或树状表格，如果数据分成两栏或更多栏，则请使用表格视图。列表视图中呈现数据的不同显示方式见图 E-14。

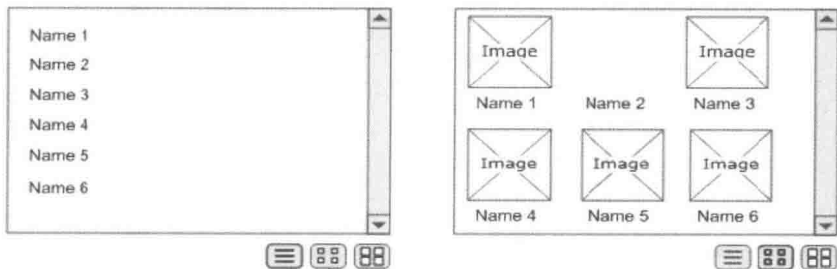


图 E-14 列表视图中呈现数据的不同显示方式

这种模式包括以下四种基本变化。

#### 1. 下拉列表 (Drop Down List)

当仅需要名称的单一列表、单一选择且空间有限时，请使用下拉列表框，见图 E-15。在使用下拉列表时，应遵循微软 Windows 用户体验指南。

#### 2. 组合框 (Combo Box)

当不仅需要下拉列表功能而且还需要能够编辑下拉列表框时，请使用组合框。使用此选项时，应遵循微软 Windows 用户体验指南，包括如何在下拉列表和组合框之间选择。

微软 Office 2007 中的字体选择器是一个组合框示例 (图 E-16)。



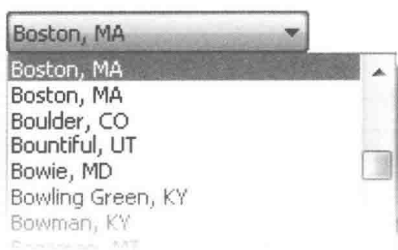


图 E-15 下拉列表

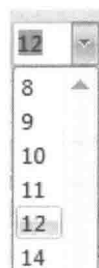


图 E-16 组合框示例

Locations defined in this project :

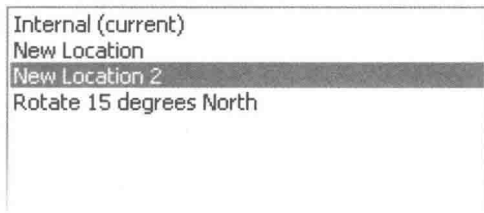


图 E-17 列表框

### 3. 列表框 (List Box)

当仅需要一个名称列表, 而 UI 上有足够空间来显示列表框时, 使用列表框, 见图 E-17。这有利于用户看到所有项目。列表框还用于需要选择多个选项的场合。使用列表框亦应遵循微软 Windows 用户体验指南。

### 4. 列表视图 (List View)

当数据包含列表、详细信息和/或图形缩略图预览选项 (如 Windows 的“打开”对话框)

时使用, 见图 E-18。

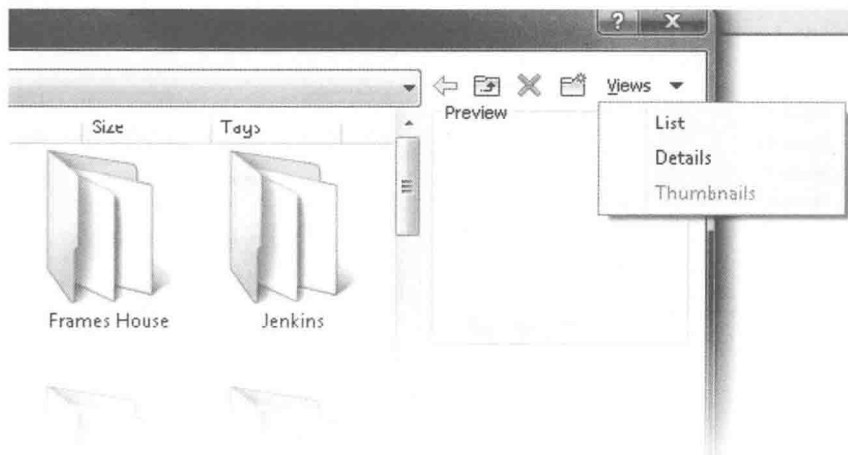


图 E-18 列表视图

### 5. 表格视图 (Table View)

用户经常需要查看集合的内容, 以便他们立刻可以容易地了解并比较多个项目的属性。为了适应这种情况, 以这种利于浏览的方式向用户显示一个格式化的表格。

### 6. 示例 (Examples)

图 E-19 是一个格式良好的表格示例。

注意: 标题单元格有别于数据单元格, 对齐方式也不同, 使它便于浏览数据。



Annual Design Conditions				
Threshold (%)	Cooling		Heating	
	Dry Build (°F)	MCWE (°F)	Dry Build (°F)	MCWE (°F)
0.1%	92.3	77.5	-6.0	-7.8
0.2%	90.3	72.3	-3.6	-5.3
0.4%	88.9	75.4	-1.3	-1.9
1.0%	88.5	73.5	2.7	-0.2
0.4%	88.9	75.4	-1.3	-1.9

图 E-19 好的表格示例

图 E-20 是一个格式欠妥的表格示例。

注意：标题单元格与数据单元格的对齐方式未做区别，不便于浏览数据。

Annual Design Conditions				
Threshold (%)	Cooling		Heating	
	Dry Build (°F)	MCWE (°F)	Dry Build (°F)	MCWE (°F)
0.1%	102.3	77.5	-6.0	-7.8
0.2%	90.3	72.3	-3.6	-5.3
0.4%	88.9	75.4	-1.3	-1.9
11.0%	115.5	73.5	2.7	-0.2

图 E-20 设计欠妥的表格示例

#### E.8.6 表格标题和表头单元格 (Table Title and Header Cells)

- 突出显示和加粗表标题，以区别于数据单元格和表头单元格
- 对于可以排序的列，单击表头对集合进行排序。表中的行要区分彼此，可用不同的颜色作为每两行中第二行的背景色。使两种颜色之间的色差保持最小以获得柔和的感觉。颜色值应相近、低饱和度——比另一行稍微暗或亮一些。通常两种颜色之一就采用页面本身的背景色。
- 确保颜色与表头和行标题有所区别。
- 标题和表头应首字符大写。

#### E.8.7 包含数值数据的列 (Columns Containing Numeric Data)

- 右对齐数据列的列标题。
- 右对齐（小数点）数值列中的数据。
- 格式化百分比列中的值，在数值右边直接加百分比符号，以确保用户知道它是百分比值。

注意：人们可能很容易忘记，他们正在查看百分比，因此在这里冗余百分比符号很重要，尤其是对于有多值的表格。

-1.23	(1.23)
0.03	0.03
-111.23	(111.47)

#### E.8.8 正负数值数据混合列 (Columns with a Mix of Positive and Negative Numeric Data)

对齐数据以使小数点对齐，见图 E-21。

图 E-21 正确对齐数值数据



### E.8.9 仅包含单个字母或控件的列(如复选框)[Columns Containing Only Single Letter or Control (Such as Check Box)]

- 将该列数据或复选符号居中。
- 将该列标题居中。

### E.8.10 不表示数字或日期的文本列 (Columns with Text that Does not Express Numbers or Dates)

- 左对齐号码列的列标题。
- 左对齐文本数据。
- 左对齐不表示数字如产品 ID 或注册号的数据。

### E.8.11 包含日期的列(视日期为文本)[Columns Containing Dates (Treat Dates as Text)]

- 左对齐日期列的列标题。
- 左对齐日期。
- 如果要呈现给国际用户, 为了避免混淆, 请在列标题中包含日期格式。

Column header 1 ▼	Column header 2 ▲
a	3
b	2
c	1

图 E-22 列排序器

### E.8.12 列排序器 (Column Sorter)

当用户在浏览如大型表的集合时, 为了找到感兴趣的值, 可能要跨越多个页面, 则可以使用列排序器, 见图 E-22。

有多种有意义的表格排序方法, 如可以动态地对其列值排序进行改变, 用户会对这样的排序方法

感到满意。

- 允许用户通过单击列标题对项目集合进行排序。
- 当用户单击列标签, 表格按该列进行排序。
- 再次单击则颠倒顺序, 这应使用向上或向下指示箭头表示。
- 哪个列可以点击、哪个是当前活动的, 应确保它是可见的。

### E.8.13 树状视图 (Tree View)

用户可能经常需要了解项目层次结构内的复杂关系, 通常这可以在“树状视图”内获得最佳显示。用户可能还需要选择一个或多个项目。如果集合是一个平面列表, 使用列表视图, 如果数据分为两列或更多列, 则使用表格视图或树状表格。

树状 UI 遵循用户启动渐进展开的原则。使用“树”允许复杂的分层数据以一个简单的, 但逐渐复杂的方式显示, 见图 E-23。如果数据变得太宽或深, 应考虑一个搜索框。

#### E.8.13.1 树状表格 (Tree Table)

就树状视图来说, 用户可能带着选择一个或多个项目的意图, 需要查看和浏览分层组织的项目集合。然而, 用户还需要看到更多的项目属性而不仅仅是名称。为了适应这种情况, 可提交用户一棵嵌入表格内的“树”。每一行都表示项目的附加属性, 展开一个节点展示另一行, 见图 E-24。

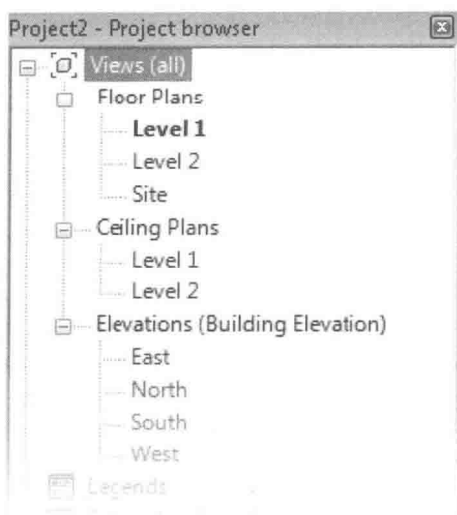


图 E-23 Revit 项目浏览器树状视图是一个很好的例子

Visibility	Projection/Surface	
	Lines	Patterns
<input checked="" type="checkbox"/> Areas		
<input checked="" type="checkbox"/> Casework		
<input checked="" type="checkbox"/> Hidden Lines		
<input checked="" type="checkbox"/> Ceilings		
<input checked="" type="checkbox"/> Common Edges		
<input checked="" type="checkbox"/> Hidden Lines	Override...	
<input checked="" type="checkbox"/> Columns		
<input checked="" type="checkbox"/> Curtain Panels		

图 E-24 Revit 可视化/图形对话框是个树表集合的搜索/过滤的好例子

当用户查看有很多项目的集合时，他们可能需要过滤项目的数量。要做到这一点，提供一种方式供用户在系统提供的过滤标准列表和用户可创建标准之间进行选择。自动选择条件过滤集合。两种最常见的方式已在 Revit 材料对话框中演示。

- 搜索框允许基于关键字过滤列表。
- 下拉列表允许基于一组定义的条件过滤列表。

#### E.8.13.2 集合编辑器 (Collection Editor)

除了查看项目集合，通常用户也会有编辑集合的需求。这可以通过关联的工具栏来完成编辑、创建、复制、重命名和删除项目。

#### E.8.13.3 编辑栏 (The Edit Bar)

按钮应从左到右按以下顺序排序，并使用以下工具提示标签：编辑、新建、复制、删除、重命名（见图 E-25）。如果编辑栏中的一个或多个按钮在某个功能中无需使用，则将其余按钮向左移动。编辑、新建、复制、删除、重命名 API 操作命令使用说明见表 E-9。

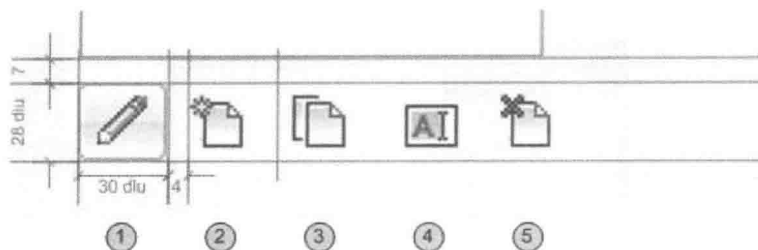


图 E-25 编辑栏

表 E-9

编辑栏 API 操作命令使用说明

	操作	使用 说明
1	编辑	如果一个项目可编辑,则可使用 Edit。如果要编辑的属性少于三个,则编辑一个项可能会启动一个单独的对话框。有关显示集合项目的详细信息,请参阅 E.8.5.5 节集合浏览器部分
2	新建	如果应用程序创建新项,则使用 New
3	复制	如果某功能只能复制现有项,则使用 Duplicate
4	重命名	如果某项允许重命名,则使用 Rename
5	删除	使用 Delete 来删除某功能

首先确保 UI 基本元素放置在布局路径中,在放置管理控件时,应遵循以下规则:

- 导航列表是主要任务:位于列表控件左下方。
- 管理列表是主要任务:位于列表控件左上方。
- 当管理的主要集合用一个组合框来表示:放到组合框右侧。

#### E.8.13.4 添加/删除 (Add/Remove)

编辑栏上的一个微小的变化是使用添加和删除按钮,以加号和减号图标表示,见图 E-26。当要向模型中的现有项目添加数据时使用添加和删除。

以下是一个很好的集合编辑器对话框例子(图 E-26),编辑栏的两种形式都有使用。添加(+)和删除(-)按钮用于将值添加到一个已有“需求因素类型”。

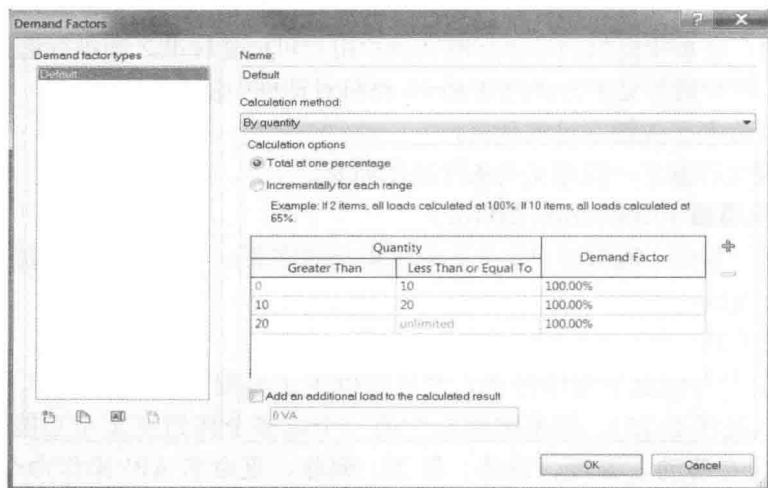


图 E-26 Revit 2011 中的需求因素对话框



### E.8.13.5 列表生成器 (List Builder)

当用户需要从一个列表添加/删除许多项目到另一个列表时, 使用列表生成器。这通常用于需要从一个现有列表 (位于左侧) 将项目添加到另一个列表 (位于右侧)。提供一个列表框视图, 包含两个列表及列表之间的两个按钮控件, 一个用于添加项、一个用于删除项, 见图 E-27。

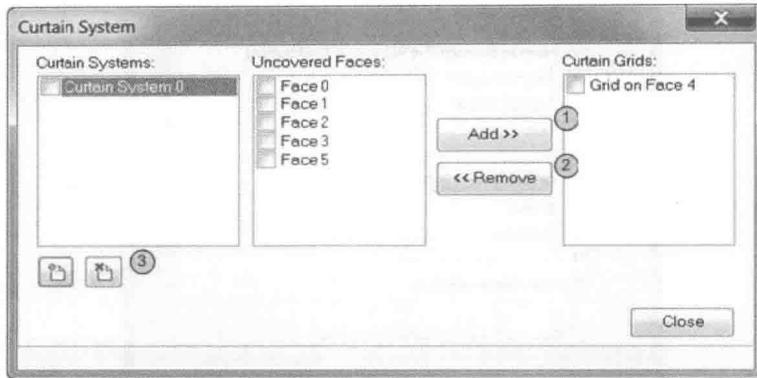


图 E-27 幕墙系统 SDK 示例

### E.8.13.6 受支持的操作 (Supported Behaviors)

列表生成器受支持的操作一览见表 E-10。

表 E-10 列表生成器受支持的操作一览表

	行为	说 明	是否必需
1	Add (到列表)	从列表 A 取出一项并添加到列表 B	是
2	Remove (从列表)	从列表 B 删除项目	是
3	集合编辑器	如果在此上下文中操控列表 A, 则使用 Collection Manager	否

根据功能, 列表生成器可以用以下两种方式之一来操作:

- 只能添加一次项目, 列表 A 中的项目只能一次性添加到列表 B。在这种情况下, 当从列表 A 中选择先前添加的项目时, 应禁用**添加**按钮。
- 可以多次添加项目。在这种情况下, **添加**按钮未被禁用, 用户可以将列表 A 中的项目添加到列表 B 多次 (次数由功能决定)。
- 如果用户需要在集合中向上或向下任意移动项目, 请在列表旁边提供**向上/向下**按钮, 见图 E-28。

### E.8.13.7 任务对话框 (Task Dialog)

任务对话框是一种模态对话框。它们有共同的一组控件, 这些控件按一个标准顺序排列以保证一致的外观和感觉。

以下情况使用任务对话框, 当系统需要:

- 为用户提供信息。
- 向用户提问。
- 允许用户选择选项来执行一个命令或任务。

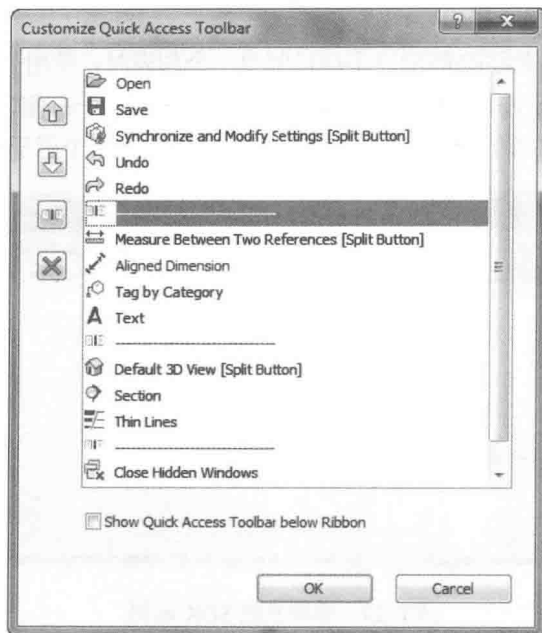


图 E-28 向上/向下按钮

图 E-29 显示了一个任务对话框模型, 包含所有可能启用的控件。大多数控件是可选的, 不需要创建一个面面俱到的任务对话框。下图样例用来简单说明任务对话框的各个部分。

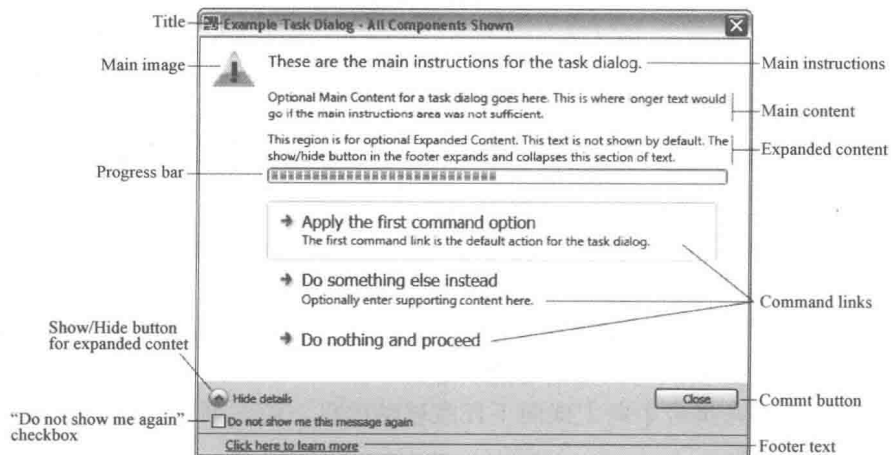


图 E-29 任务对话框中可见的所有组件

注意: 在实际应用中不会出现这种特别的任务对话框。只有一小部分会被同时使用。任务对话框无法显示其他控件, 例如, 文本输入框、列表框、组合框、复选框等。它们也只能提供单步、单行为操作; 这意味着用户可以做出单一选择并完成任务对话操作。因此, 需要这些附加控件或多步骤操作作为向导的任何对话, 并非使用任务对话框的合适对象。它们需要使用 .NET 控件, 来实现外观和感觉与任务对话框相



似的自定义对话框。

为了与 Autodesk 其他产品保持一致,何时、何处及如何使用各个任务对话框组件,将在下部分讲解。

### 1. 一般性设计原则 (General Design Principles)

以下是一些可应用于全球各种语言任务对话框的指导性原则。

检查任务对话框中内容,看一看:

- 为进行有信息根据的操作,它提供所需的全部信息了吗?
- 信息是否过于充斥技术化、术语化,目标用户能理解吗?

### 2. 以下要点适用于产品的英文版本: (The Following Points Apply to English Language Versions of Product Releases:)

- 文本应按句法格式编写,规范文本大小写和标点符号。而标题和命令按钮文本需用标题格式编写。
- 标点后应放置一个单独空格。例如,不要在句子末尾的句点后放置两个空格。应避免使用括号来处理复数名词,而代之以改写句子。例如:
- 用 "At least one referenced drawing contains one or more objects that were created in..." 代替 "The referenced drawing (s) contains object (s) that were created in ...".
- 对话框中调用任何第三方应用程序后都应标明版权符号©。

### 3. 标题 (必需的) [Title (Required)]

所有任务对话框都需要一个标题。任务对话框的标题应该是描述性的,并尽可能唯一。

任务对话框标题应采用以下格式:

<功能名称>—<短标题>

- 其中<功能名称>是触发任务对话的模块。
- <短标题>是导致对话显示的行为。
- 例如:
  - 参考编辑—版本冲突。
  - 图层—删除。
  - 材料清单—编辑公式。

如果可能,标题的第二个部分<短标题>使用动词,如创建、删除、重命名、选择等。在没有明显适用的功能名称(或几个)的情况下,短标题也足够了。

任务对话框的标题不应该是产品名称,如 AutoCAD Architecture。

### 4. 标题栏图标 (Title Bar Icon)

出现在标题栏最左边的图标应为主体应用程序的图标——包括第三方插件。任务对话框标题中可能包含插件的名称以指定消息来源,但任务对话框中所有的视觉品牌应该与主体应用程序匹配,如 Revit Structure、Inventor、AutoCAD Electrical 等。

### 5. 主要说明 (必需的) [Main Instructions (Required)]

这是显示在任务对话框顶部的大段主要文本,见图 E-30。

- 每个任务对话框都应该有某种主要说明。
- 文本应不超过三行。





- [英文版] 主要说明应以句子编写, 规范文本大小写及标点符号。
- [英文版] 直接用 “you” 称呼用户。
- [英文版] 当面对多个命令链接选项时, 标准的主要说明的最后一行应是: “What do you want to do?”。

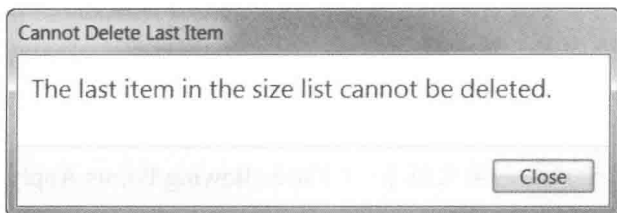


图 E-30 仅有一段主要说明文本、非常简单的任务对话框

#### 6. 主要内容 (常用可选项) [Main Content (Optional - Commonly Used)]

它是在主要说明下方显示的较小字体的文本, 见图 E-31。

- 主要内容是可选的。它主要用于任务对话所需的说明不适合全部显示在主要说明区的情况。
- 主要内容不应只是以不同方式重述主要说明, 它应该包含以主要说明为基础的额外信息或强化主要说明。
- [英文版] 主要说明应按句法格式编写 (正常大写及标点符号)。
- [英文版] 需要时, 直接用 “you” 称呼用户。

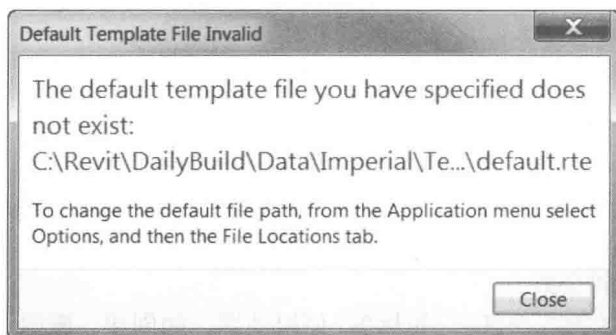


图 E-31 主要说明和主要内容两者都有的任务对话框

#### 7. 扩展内容 (极少使用的选项) [Expanded Content (Optional - Rarely Used)]

此文本默认隐藏, 当按下显示按钮时才显示在任务对话框底部, 见图 E-32。

- 扩展内容是可选的, 不常使用。它用于没有必要预先显示额外信息, 或大多数情况下并不使用此信息的场合。
- [英文版] 主要说明应按句法格式编写 (正常大写及标点符号)。
- [英文版] 需要时, 直接用 “you” 称呼用户。

#### 8. 主图像 (较少使用的选项) [Main Image (Optional - Low Usage)]

任务对话框支持在主要说明左侧含有图像。在任务对话框之前, 大多数对话框通常都



有某种图标，表示它含有某种有用信息，如警告和错误等。

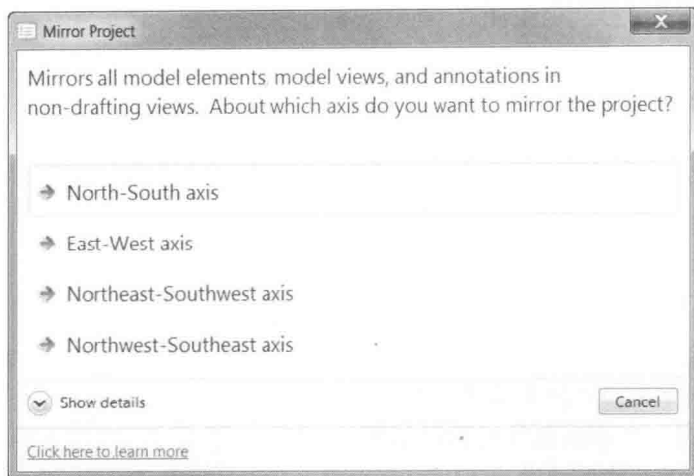


图 E-32 “显示详细信息”按钮显示主要内容的附加文本

由于图像使用频繁，因此对话框中任何图像的使用价值都很低。

对于 Autodesk 产品，警告图标（黄色三角形中的惊叹号）仅用于某个操作可能导致某种程度上毁灭性错误的情形，如导致数据丢失或重大返工时间损失。

一些例子包括：

- 覆盖文件。
- 保存为一个旧的或不同的格式文件，可能会丢失数据。
- 永久性删除数据。
- 移动或重命名而阻断文件之间的关联。

这只是部分列表，使用主图像时应避免这种情形的异常。参见图 E-33 示例任务对话框警告图标。

9. “不再显示”(DNSM)复选框(可选项)[“Do not Show Me Again”(DNSM)Checkbox (Optional)]

任务对话框支持“Do not show me again”复选框（图 E-33），可在对话框中启用它，用户可以选择不再看此消息。在英文版本中此复选框标签的标准措辞是：“Do not show me this message again”，这里“Do not”不是“Don't”的缩写，句末也没有标点符号。

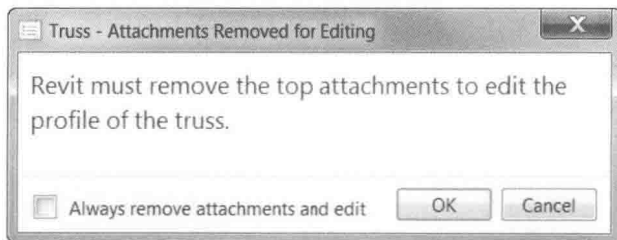


图 E-33 DNSMA 复选框作为“Always……”单选框示例



对于单一动作，其操作应该是“总是<动作>”，例如：

- 如果动作是“保存当前绘图”，则复选框标签显示为“总是保存当前绘图”。
- 如果动作是“对象转换为线框”，则复选框标签显示为“总是转换对象为线框”。

在有多项选择的情况下：

- 应使用一般措辞“始终执行当前选择”。
- 应使用命令链接以显示可选项。如果使用了包含一个**取消**按钮的一组按钮，看起来“取消”是一个以后始终可执行的选项。

#### 10. 脚注文本 (可选项) [Footer Text (Optional)]

脚注文本用于链接帮助文本。它取代了以往对话框中的**帮助**或“?”按钮，并链接到现有帮助链接的同一位置。英文版本中的脚注文本应为：“Click here to learn more”，文本应按句法格式写成陈述句，但不要加句点。

图 E-31 是带脚注文本的任务对话框示例。

#### 11. 进度条 (极少使用的选项) [Progress Bar (Optional - Rarely Used)]

在任务对话框需要显示进程的实例中，或处理一个可选项可能需要数秒或更多时间时，可使用进度条。

#### 12. 命令链接 (常用可选项) [Command Links (Optional - Commonly Used)]

任务对话框中，用户选择操作有两种方法：命令链接和提交按钮，见图 E-34 和图 E-35。

- 多个可选项 (应避免只显示一个命令链接)。
- 短文本将有助于用户确定最佳选择。

命令链接处理方案如下：

- 执行 A、B 或 C。
- 执行 A、B 或 A 和 B。
- 执行 A 或不执行 A，等等。

命令链接文本有两部分：

(1) 主要内容：它是在一行中写完的陈述句，是任何命令链接都需要的。对于英文版本，请用没有句点的句法格式写。

(2) 补充内容：它是可选的补充文本，目的是让主要内容更清晰易懂。对于英文版本，应按带句点的正常格式写。

在顶部的第一个命令链接是任务对话框的默认操作。如果实质上没有比其他常见用例更合适的选择，它应该是最常用的操作或潜在破坏性最小的操作。

#### 13. 提交按钮 (常用可选项) [Commit Buttons (Optional - Commonly Used)]

提交按钮是任务对话框中脚注部分的简单按钮。标准英文术语包括：

- OK。
- Cancel。
- Yes。
- No。
- Retry。

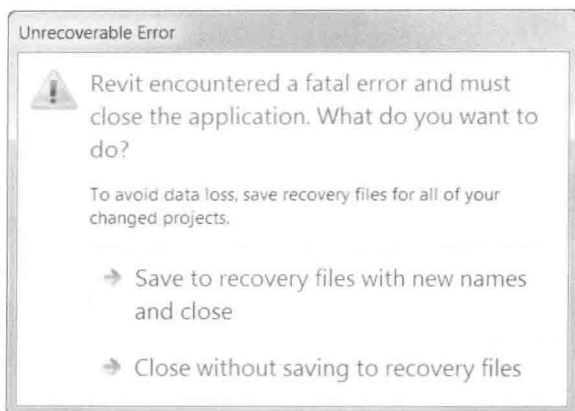


图 E-34 有两个命令链接的任务对话框

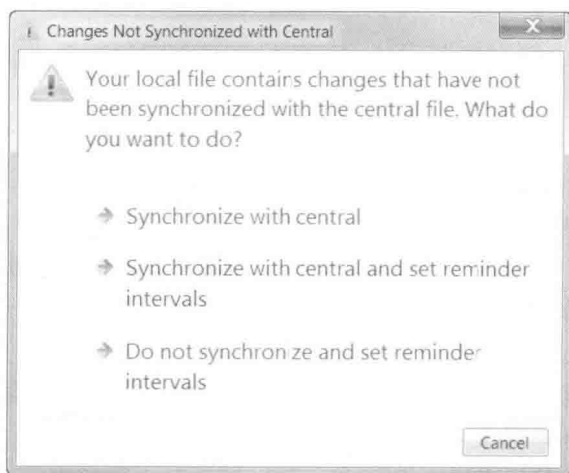


图 E-35 有命令链接和一个命令按钮的任务对话框

- Close。

提交按钮也可以使用自定义文本，但不推荐。

每个主要按钮类型的正确用法说明：

- **确定按钮**，仅用于任务对话框所提出的问题可以用“OK”来回答的情况下。
- **取消按钮**，仅用于一个任务真正可以被取消，这意味着任务对话框所触发的操作将被终止，并且不会提交任何更改。它可以与其他提交按钮或命令链接组合使用。
- **是和否按钮**，总是以组合方式使用，主要说明和/或主要内容中的文本可以“Yes/No”来结束。
- **重试按钮**，必须至少有一个作为备用选项的**取消按钮**一起显示，因而用户可以选择取消重试。
- **关闭按钮**，用于任何纯信息任务对话框，即用户没有任何其他操作可选择，而仅可阅读文本并关闭对话框。原先这类对话框经常会使用**确定按钮**。在任务对话框里，这种情况还请使用**关闭按钮**。



以下是一些如何使用提交按钮的示例：

- 图 E-29 是一个带**关闭**按钮的纯信息任务对话框的例子。
- 图 E-31 是一个带命令链接的**取消**按钮的例子。
- 图 E-32 是一个有**确定**和**取消**按钮的任务对话框的例子。

### E.8.13.8 默认按钮或链接 (Default Button or Link)

所有任务对话框都应明确指定默认按钮或链接。如果任务对话框有一个**确定**按钮，则它应该是默认按钮。

请注意，唯一例外的是带有命令链接的自定义任务对话框，这些命令链接的操作也同样可行，若没有其他“更好的”操作，则不应该指定默认选项。而仅使用提交按钮的所有对话框则必须指定一个默认按钮。

## E.8.14 导航 (Navigation)

### E.8.14.1 选项卡 (Tabs)

选项卡在选项之间有松散联系时使用，同时要求不同的信息“块”必须存在于同一界面内，但却没有足够的空间来清晰地显示全部内容。

把 UI 分开成不同的模态区，每个区都用带有描述性标签的“选项卡”表示。整个对话框应视为有一组单一提交按钮的单一窗口。

- 所有标签都应同时可见。
- 静态 UI 中不会仅有一个单一选项卡，如对话框。因而要为页面或对话框使用选项卡标题。例外情况：如果选项卡数量会动态增加，则默认为一个，如 Excel 打开工作簿选项卡。
- 选项卡仅起导航作用。除了简单地切换窗口中的页面，选择某个选项卡时不应执行任何其他操作（如提交）。
- 避免在选项卡式窗口内嵌套选项卡。在这种情况下应考虑启动子窗口。
- 不要更改选项卡上基于在父窗口内交互的动态标签。

### E.8.14.2 选项卡变种 (Variations)

#### 1. 变种 A：水平选项卡 (Variation A: Horizontal Tabs)

水平选项卡见图 E-36。

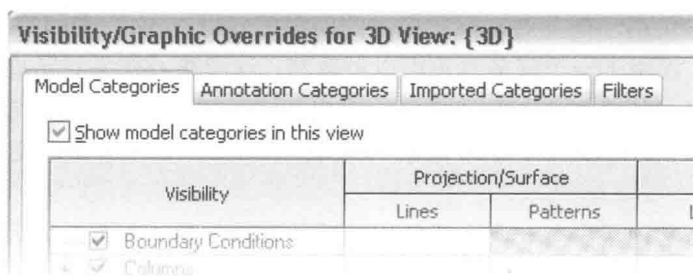


图 E-36 水平选项卡

避免多行水平选项卡。如果需要第二行，应考虑垂直选项卡。



## 2. 变种 B: 垂直选项卡 (Variation B: Vertical Tabs)

垂直选项卡见图 E-37。

垂直选项卡用于：避免选项卡从左到右的布局过程中，因选项卡较多，迫使水平布局另起第二行的情形。

### E.8.14.3 键盘辅助功能 (Keyboard Accessibility)

#### 1. 选项卡顺序 (Tab Order)

按 **Tab** 键在对话框中每个可编辑控件焦点之间循环。一般规则是从左到右、自上而下。

(1) 默认 **Tab** 定位点位于对话框左上角第一个控件。

(2) 向右移动，直到当前行中最后一个控件。

(3) 移动到下一行，并开始从最左侧的控件，向右移动。

(4) 重复步骤 2，直到最后一行。最后总是结束于 **OK/Cancel/Apply** 行。

- 左、右箭头，上、下箭头，**Tab** 和 **Shift-Tab** 键分别都有同样作用。除了焦点在下列控件上：
  - 列表控件或组合框：分别地，左/右、上/下箭头移动光标在列表中上/下。
  - 网格控件：分别地，左/右移动光标左/右跨列，上/下箭头移动光标在列表中上/下。
  - 滑块：左/右、上/下箭头分别左/右移动滑块。
  - 下拉列表：左/右、上/下箭头分别在下拉列表中上/下移动。
- 从概念上来说，每个组都可视为嵌套的对话框，首先，在每个组内遵循上述规则，从左上方那组开始移动，向右移到最右侧不再遇到更多的组，然后移到下一行。
- 如果是选项卡式对话框，默认 **Tab** 定位点应为默认选项卡。

提示：通过切换选项卡顺序可视化助手，**Visual Studio** 可以协助创建和编辑选项卡顺序（由“查看” ➤ “选项卡顺序”菜单访问）。

#### 2. 快捷键 (Access Keys)

- 对话框的每个可编辑控件都应该有唯一的快捷键字母（由控件标签中带下划线的字母表示）。
- 用户按下 **Alt** 键和指定的快捷键则控件被激活，等同于控件被点击。
- 默认按钮不需要快捷键，因为它已被映射到回车键。
- 取消或关闭按钮也不需要快捷键，因为 **Esc** 键映射到它。有关详细信息，请参阅 E.8.14.4 节提交更改。

#### 3. 显示更多按钮 (Show More Button)

遵循渐进式展开的原则，用户可能需要一种方式，以获得比用户界面中默认的显示内容更多的数据。“显示更多”按钮通常有两种实现方式：

扩充器按钮：提供一个有标签的按钮，如“<预览”或“显示更多>”。双尖括号“>”应指向将要显示的新信息窗格。当窗格打开时，双尖括号应切换到指示该窗格应如何“关闭”。

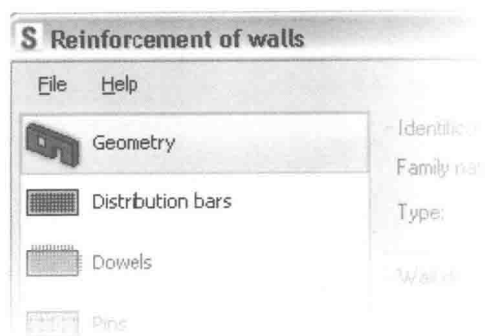


图 E-37 垂直选项卡

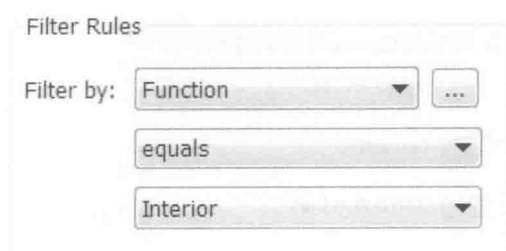


图 E-38 Revit 视图过滤器对话框中的对话框启动器按钮

参见图 E-13—— Revit 材料对话框。

对话框启动器：一个带省略号 “...” 的按钮，它启动一个独立对话框，见图 E-38。这通常用于提供一个单独的用户界面来编辑所选项。

E.8.14.4 提交更改 (Committing Changes)

模态对话框用于对项目文件内的数据进行更改。当模态对话框或窗体中有一个编辑器，且一系列编辑已排队等待并需要立刻提交时，

使用提交更改。如果对话性质是纯粹的信息，请使用任务对话框，它有它自己的提交规则。

每个模态对话框或 Web 窗体都必须有一组提交按钮，以提交更改和/或取消任务、和/或关闭对话框。

E.8.14.5 尺寸 (Sizing)

提交按钮的大小见图 E-39。

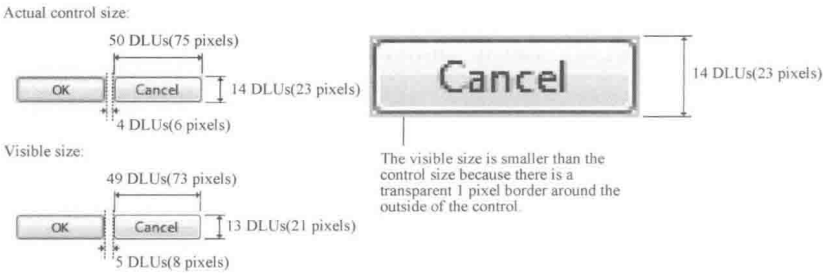


图 E-39 提交按钮的大小 (摘自微软 Windows 用户体验指南)

E.8.14.6 布局 (Layout)

不同窗口类型的提交按钮样式综述见表 E-11，按钮类型说明见表 E-12。

表 E-11 不同窗口类型的提交按钮样式综述

模式	提交按钮样式
模态对话框	确定/取消或 [操作] /取消
非模态对话框	对话框和标题栏的关闭按钮
进度显示器	如果返回到先前的环境状态则使用取消按钮 (不留任何意外结果)； 否则，使用中止按钮

表 E-12 按钮类型

1	默认 (确定或其他操作) 按钮
2	取消或关闭按钮
3	应用按钮
4	对话框按钮 (可选)



提交按钮应遵循这种布局模式，见图 E-40。

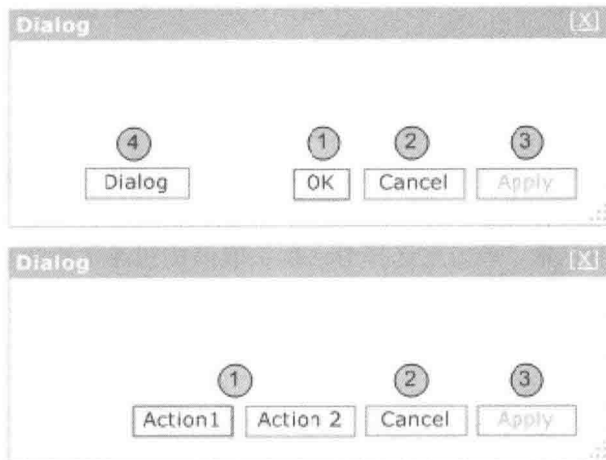


图 E-40 标准的提交按钮布局

按此顺序并右对齐放置**默认**、**取消**以及**应用**按钮。**对话框**按钮（如果有的话）则左对齐，而**帮助**按钮（如果有的话）右对齐。

#### E.8.14.7 默认（确定或其他动作）按钮 [ Default (OK and other Action) buttons ]

对话必须有一个默认的操作按钮。这个按钮应紧密映射到对话的首要任务。它可以是确定按钮，或是对操作更具描述性的动词按钮。见表 E-11。

- 用不会带来破坏性结果的按钮作为默认按钮。
- 默认按钮的快捷键是回车键。

#### E.8.14.8 确定按钮 (OK button)

确定按钮用于保存一个或多个设置。确定按钮的规则如下：

- 确定按钮用于对话框中唯一可以提交的操作（除“取消”之外）。不要将确定按钮与其他操作按钮混合使用。
- 在模态对话框中，单击确定就意味着应用该值、执行任务，且不使用确定按钮回答问题就关闭窗口。
- 正确地标记确定按钮，应该标记为“OK”，而不是“Ok”或“Okey”。
- 不要在非模态对话框中使用确定按钮。而应使用操作按钮或关闭按钮。

#### E.8.14.9 动作按钮 (Action buttons)

动作按钮的描述性动词由设计者定义。动作按钮规则：

- 动作按钮可以用来更清晰地描述在它被单击时将采取的行动。
- 必须设置一个默认动作按钮，且应该是对话中与首要任务有最密切映射关系的动作。
- 可以有一个或多个动作按钮，但不要混合使用确定按钮和动作按钮。
- 使用**取消**或**关闭**按钮作为否定性提交按钮，取代对主要说明的具体响应。
- 否则，如果用户想取消，否定性提交需要为此特定的小任务作更多的思考。





#### E.8.14.10 取消或关闭按钮 (Cancel or Close Button)

- 标题栏上的**关闭按钮**的作用等同于对话框中的**关闭或取消按钮**。
- Esc 是**取消或关闭按钮**的键盘快捷键。

#### E.8.14.11 取消按钮 (Cancel Button)

- **取消按钮**仅用于中止任务且没有更改要提交。
- 单击**取消按钮**意味着放弃所有更改、取消任务、关闭窗口，回到先前的环境状态，不会留下任何意外结果。
- 对于嵌套选择对话框，单击主选择对话框中**取消按钮**通常意味着放弃自身拥有的选择对话框所做的任何更改。
- 不要在非模态对话框中使用取消按钮，而应使用关闭按钮。

#### E.8.14.12 关闭按钮 (Close Button)

- **关闭按钮**用于非模态对话框，同样还用于模态对话框无法使用“取消”的情况。
- 单击**关闭按钮**意味着关闭该对话框窗口，并保留现有的任何更改。

#### E.8.14.13 应用按钮 (可选) [Apply Button (Optional)]

应用按钮将提交对话框内所有选项卡、页面或层次级别中所做的任何更改，而不关闭对话框。最理想的情况是，用户可看到所提交变化的反馈信息。以下是应用按钮的一些基本规则：

- 在模态或非模态对话框中，单击应用都意味着应用该值、执行任务，而不关闭窗口。
- 在非模态对话框中，仅当这些任务执行前需要大量或未知的预先准备时间时，才使用应用按钮，否则数据更改应立即执行。
- 在没有做出任何更改的情况下，应用按钮为禁用状态。在做出更改后，它变为可启用状态。
- 单击**取消按钮**不会撤销已由**应用按钮**提交的任何更改。
- 与子对话框互动（如确认）不会导致应用功能变为可启用状态。
- 在提交子对话框后（如确认消息）单击**应用按钮**，会触发确认先前做出的所有更改。

#### E.8.14.14 对话框按钮 (可选) [Dialog Button (Optional)]

对话框按钮执行其本身动作。例子包括：在打开的对话框中用于管理收藏夹的重置和工具。它们应该与对话框的左侧对齐（帮助按钮则右对齐，如果有的话），并且绝不要将它们设为默认按钮。

#### E.8.14.15 “实现”说明 (Implementation Notes)

- 键盘快捷方式：每个提交按钮应该有一个映射到它的快捷键。默认按钮应映射到回车键。
- 关闭按钮（无论是取消还是关闭）应映射到 Esc 键。
- 如果有应用按钮，且不是默认按钮，则它应映射到 Alt-A。

## E.9 功能区指南 (Ribbon Guidelines)

以下是 API 开发人员可以修改的功能区用户界面方面的内容。必须遵循这些指南，使应用程序用户界面符合 Autodesk 标准。



### E.9.1 功能区选项卡布置 (Ribbon Tab Placement)

要在功能区上腾出更多空间，第三方应用程序才可以添加分析选项卡以及插件选项卡的功能区控件。

- 添加和/或修改 Revit 图元的应用程序应该添加到插件选项卡。
- 分析 Revit 模型中现有数据的应用程序应添加到分析选项卡。
- 应用程序绝不能既添加到插件选项卡又添加到分析选项卡。

### E.9.2 上下文选项卡焦点的用户选项 (Contextual Tab Focus User Option)

Revit 2012 产品线包含一个用户选项 (位于选项对话框的用户界面选项卡)，允许用户选择是否在选择后自动切换到上下文选项卡。在默认情况下，此选项设置为自动切换。对于某些 API 应用程序来说，禁用此选项可能更为有利，以避免用户从插件或分析选项卡中切换出来。在这些情况下，最好是将文件中的这些选项告知用户和/或作为安装程序用户界面中的信息文本。

### E.9.3 每个选项卡的面板数量 (Number of Panels per Tab)

每个 API 应用程序应该只有一个面板添加到插件选项卡。

### E.9.4 面板布局 (Panel Layout)

下述内容定义了插件选项卡上布置面板的正确方法。可仿效图 E-41 所示总体布局下的面板示例。

### E.9.5 总体布局 (General Layout)

面板应该有一个大按钮作为最左侧的控件，见图 E-41。这个按钮应该是应用程序中最常访问的命令。当面板隐缩时 (请参阅 E.9.7 节面板大小调整和隐缩)，最左边的按钮图标将代表整个面板。此按钮可能是按钮组中唯一的按钮，也可能后随一个大按钮和/或小按钮堆。

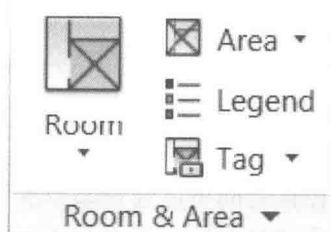


图 E-41 Revit 2011 产品中房间和面积面板

面板不应超过三列。如果需要多个控件，请使用下拉按钮。

面板应该只包含启动命令和控制应用程序的控件。管理设置或启动帮助及“关于此应用程序”控件应布置在滑出面板上。

### E.9.6 小按钮堆 (Small Button Stack)

- 堆必须至少有两个按钮并不得超过 3 个。
- 小按钮的顺序应遵循最常用的在底部、最少用的在顶部。这是因为访问更频繁的命令应更接近建模窗口。

### E.9.7 面板大小调整和隐缩 (Panel Resizing and Collapsing)

默认情况下，面板将基于客户安装顺序从左到右按降序布置。一旦组合面板的宽度超过当前窗口的宽度，面板将按以下顺序从右起开始调整大小：



(1) 大按钮面板。

1) 小按钮失去其标签, 然后:

2) 面板隐缩为单个大按钮 (代表面板的图标为左边第一个图标)。

(2) 仅有小按钮堆的面板。

1) 小按钮失去其标签, 且面板标签被截短为一行有 4 个字符及 3 个点的省略号。

2) 如果小按钮堆是面板最左边的控件, 那么顶部按钮必须有一个与其关联的大图标。

在隐缩时, 此图标将代表面板。

关于按钮/链接应设在主用户界面上, 而不是功能区面板上。

注意: 功能区组件会自动处理面板大小调整和隐缩。

## E.9.8 功能区控件 (Ribbon Controls)

### E.9.8.1 功能区按钮 (Ribbon Button)

功能区按钮是最基本、最常用的控件。按下一个按钮调用一个命令。

功能区按钮有 3 种尺寸:

- 大按钮: 必须有一个文本标签。
- 中按钮: 可以有一个文本标签。
- 小按钮: 可以有一个文本标签。

### E.9.8.2 单选按钮 (Radio Buttons)

单选按钮组表示一组控件是互相排斥的, 每次只能选择一个。这些按钮组可以横向堆排 (如图 E-42 所示对齐按钮)。

### E.9.8.3 下拉按钮 (Drop-down Button)

- 顶部标签应充分描述下拉列表的内容。
- 列表中的每个项目应包含一个大图标。
- 控件之间可以选择添加水平分隔符。如果一个按钮下的项目是逻辑分组的, 则应使用水平分隔符以区分不同的组。

楼板下拉按钮见图 E-43。

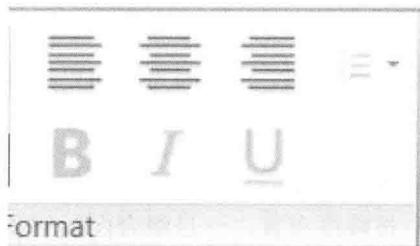


图 E-42 Revit 2011 设置文本格式面板



图 E-43 Revit Architecture 楼板下拉按钮



#### E.9.8.4 拆分按钮 (Split Button)

拆分按钮是有一个默认命令的下拉按钮，可以点击按钮左侧进行访问。按钮的右侧，用一个小的垂直分隔符分隔，打开一个下拉列表。默认命令应是列表顶部命令的副本。

拆分按钮的默认命令可以同步。也就是说，默认命令可随上次使用的下拉列表中的命令而改变。

#### E.9.8.5 组合框和文本框 (Combo Box and Text Box)

在功能区中的组合框和文本框，其使用准则和在对话框中是相同的。可参阅 E.8.3 对话框控件。

#### E.9.8.6 滑出面板 (Slide-out Panel)

滑出面板见图 E-44。

一般来说滑出应用于面板相关的命令，但并非主要的或常用的一种方式。

每个打开的面板都可选择固定打开。否则，一旦鼠标离开面板，则会自行关闭。

滑出面板的 3 个使用建议：面板内任务相关的设置对话框启动命令，帮助按钮和一个关于按钮。

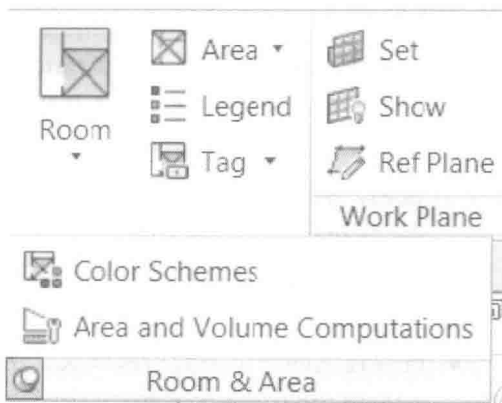


图 E-44 Revit Architecture 滑出面板

#### E.9.8.7 垂直分隔符 (Vertical Separator)

可在控件或控件组之间添加垂直分隔符，以在面板内创建不同的命令分组。一个面板内应该最多只有两个分隔符。

#### E.9.8.8 图标 (Icons)

图标应有美观、能清楚表达其功能的设计。

### E.9.9 文本用法 (Text Usage)

#### E.9.9.1 按钮标签 (Button Labels)

以下指南只用于英文标签：(其他语言亦可参照——译注)

- 不能有任何标点符号 (连字符、“&”或“/”除外)。
- 不能超过 3 个单词。
- 不能超过 36 个字符。
- 必须有标题，如 “Show Mass”。
- 必须用和号 “&” 代替 “and”。和号前后应留空格。
- 必须用正斜杠 “/” 代替 “or”。斜杠前后不应留空格。
- 只有大按钮可以有两行标签，但不能超过两行。所有其他控件的标签必须在一行上。
- 按钮标签必须不包含省略号。
- 每个词都必须大写，冠词 (“a” “an” 和 “the”)、并列连词 (如 “and” “or” “but” “so” “yet” “with” 和 “nor”) 以及少于 4 个字母的介词 (如 “in”) 除外。第一个和最后一个词总是大写的。



### E.9.9.2 面板标签 (Panel Labels)

这些指南只用于英文标签。命令标签部分的所有规则都适用于面板标签，此外：

- 面板的名称应该是具体的。含糊的、非描述和非特定性的术语用来描述面板内容会降低标签的效用。
- 应用程序不能使用缩略语 “misc.” 或 “etc” 作为面板名称。
- 面板标签不应包含术语 “add-ins”，因为它是冗余选项卡标签。
- 面板标签可以包括第三方产品或供应商名称。

### E.9.10 提示框 (Tooltips)

以下是编写提示信息文本方面的指南。在有限的显示空间内，文字要简明。

#### E.9.10.1 本地化注意事项 (Localization Considerations)

- 为每个单词计数。这对本地化提示文本特别重要。
- 不要使用动名词（动词形式用作名词），因为它们会与分词（动词形式用作形容词）混淆。例如，“Drawing controls” 中 “Drawing” 可能用作动词或名词。更好的写法是 “Controls for drawing”。
- 不要在工具提示中使用冗长的步进式过程。这些属于帮助。
- 使用惯用术语。
- 确保连词的使用不会引起歧义。例如，与其用 “replace and tighten the hinges”，不如分解为两个简单而重复的句子—— “Replace the hinges. Then tighten the hinges”。
- 注意“助”动词。助动词包括 shall、may、would have、should have、might have 以及 can。例如，can 和 may 可以译为“能力”和“可能性”。
- 注意隐式复数名词，如 “object and attribute settings”。意思是 “settings for one object and one attribute” 还是 “settings for many objects and many attributes” 呢？
- 谨慎使用那些既是名词又可以是动词的词。使用冠词或改写短语，如 “Model Display” 中的 “Model” 在软件中可以是名词或动词。另一个例子是 “empty file”，其意思可以是 “to empty a file”，也可以是 “a file with no content”。
- 谨慎使用隐喻。隐喻是很微妙的，通常在某些图标上下文中讨论，可能因人文差异或跨文化的理解不同而不恰当。文本隐喻如 “places the computer in a hibernating state” 就可能是个问题。代之以，还可以说 “places the computer in a low-power state”。

#### E.9.10.2 写法/措辞注意事项 (Writing/Wording Considerations)

- 使用简单的句子。推荐 “动词-宾语-副词” 格式。
- 使用描述具体动作的强动词和具体动词（如 “tile”），而不是弱动词（如 “use to...”）。
- 使用主动语态（例如，“Moves objects between model space and paper space”）。
- 使用描述性的风格，而不是命令式风格（“Opens an existing drawing file” 对比 “Open an existing drawing file”）。
- 通过使用第三人称单数，使工具提示说明易于辨认（例如 “Specifies the current color” 而不是 “Specify the current color”）。
- 不要使用俚语、俗语或难于理解的缩略词。



### E.9.10.3 格式编排注意事项 (Formatting Considerations)

- 句子之间只使用一个空格。
- 避免文本重复。工具提示中的内容应该是唯一的，并能提高使用价值。
- 注重提示信息的质量和易懂。描述清楚吗？有帮助吗？
- 除非是系统变量或命令，不要使用粗体。尽管亚洲语言支持粗体，但基于易读性及文体风格，仍强烈建议避免使用粗体和斜体。
- 避免多重缩写。例如，单词“Number”有许多常用缩略词：No.、Nbr、Num、Numb。最好拼写清楚。

#### 1. 好例子: (Good Example:)

好例子见图 E-45。

一个更有用的描述性语句的例子可能是“Adds a file such as a .bmp or .png”。这提供了更详细的信息并让用户能更深入地了解其功能。

#### 2. 差例子: (Good Example:)

在图 E-46 这个例子中，提示内容逐字重复提示标题，没有增加提示的使用价值。此外，如果译者无法确定此字符串到底是标题名称还是说明性文字，将很难决定翻译风格。

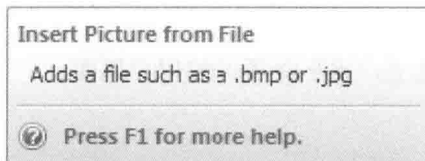


图 E-45 好例子

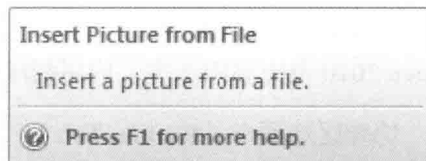


图 E-46 差例子

其他准则问题，遵循微软标题和句首字符大写准则。

### E.9.10.4 标题大写 (Title Case)

- 大写所有名词、动词（包括 is 和其他形式的 be 动词）、副词（包括 than 和 when）、形容词（包括 this 和 that）和代词（包括 its）。
- 大写第一个和最后一个词，而不管它们的词类（例如，The Text to Look For）。
- 大写作为动词短语一部分的介词（例如，Backing Up Your Disk）。
- 不要大写冠词（a、an、the），除非该冠词是标题中第一个词。
- 不要大写并列连词（and、but、for、nor、or），除非该连词是标题中第一个词。
- 不要大写 4 个以下（含 4 个）字母组成的介词，除非该介词是标题中第一个词。
- 不要大写不定式短语中的 to（例如，How to Format Your Hard Disk），除非它是标题中第一个词。
- 大写复合词中第二个单词首字母，如果它是一个名词或专有形容词、“电子词”或其他等权词（例如，E-Commerce、Cross-Reference、Pre-Microsoft Software、Read/Write Access、Run-Time）。如果第二个单词是其他词类则不要大写，如介词或其他关键词（例如，Add-in、How-to、Take-off）。
- 大写通常不会大写的用户界面和应用程序编程接口术语，除非它们区分大小写（例



如, fdisk 命令)。

- 遵循编程语言中传统的大写关键词和其他专业术语(例如, The printf function、Using the EVEN and ALIGN Directives)。
- 只大写每个列标题的第一个词。

#### E.9.10.5 句首大写 (Sentence Case)

- 总是大写新句子的第一个单词首字母。
- 冒号后面的单词不要大写, 除非这个词是一个专有名词, 或冒号后面的文本是一个完整的句子。
- 继一个长破折号之后的这个词不要大写, 即使破折号之后的文本是一个完整的句子, 除非它是一个专有名词。
- 总是大写任何句点之后新句子的第一个字。如果该句首是个区分大小写的小写单词, 则改写句子。

## E.10 通用定义 (Common Definitions)

### E.10.1 功能区 (Ribbon)

Revit 2010 及更高版本中, 应用程序框架顶部之上, 水平选项卡的用户界面。

### E.10.2 功能区选项卡 (Ribbon Tab)

功能区分成选项卡。插件功能区选项卡, 在至少安装了一个插件时才会出现, 可供第三方开发人员添加面板。

### E.10.3 功能区面板 (Ribbon Panel)

功能区选项卡分成水平分组命令。插件面板表示可用于第三方开发人员的应用程序的命令。插件面板相当于 Revit 2009 中的工具栏。

### E.10.4 功能区按钮 (Ribbon Button)

按钮用于启动命令。按钮分大、中、小 (按钮不管大、小均可为简单按钮, 亦可为下拉按钮)。

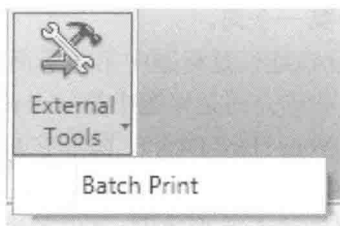


图 E-47 插件选项卡上的外部工具菜单按钮  
单按钮中。

### E.10.5 菜单按钮 (Menu Button)

插件选项卡上的默认第一个面板是外部工具面板, 它包含一个标题为“外部工具”的按钮, 见图 E-47。“外部工具”菜单按钮相当于 Revit 2009 中的“工具”>“外部工具”菜单。任何注册到.addin 清单文件中的外部命令都将出现在该菜单按钮中。

### E.10.6 下拉按钮 (Drop-down Button)

下拉按钮可展开下拉菜单以显示两个或更多命令。每个子命令都可以有自己的大图标。

E.10.7 垂直分隔符 (Vertical Separator)

垂直分隔符是在面板上的控件之间添加的细垂直线。

E.10.8 工具提示 (Tooltip)

当用户将鼠标指针悬停于功能区按钮上，会显示一个工具提示小面板。工具提示提供该命令预期行为的简要说明。

E.11 用语定义 (Terminology Definitions)

下列几个大写词用来表示标准执行的程度要求。本节定义应如何理解这几个特定词 (表 E- 13)。这些解释源自 Internet Engineering Task Force RFC 2119。

表 E-13 特定词定义

措辞	定 义
MUST	“MUST” 或 “SHALL”，意味着该条款应绝对执行
MUST NOT	“MUST NOT” 或 “SHALL NOT”，意味着该条款应绝对禁止
SHOULD	“SHOULD”或形容词“RECOMMENDED”，意味着可能在特定情况下有适当理由忽略该项条款，但必须充分理解其完整含义并仔细权衡之后，再做决定
SHOULD NOT	“SHOULD NOT” 或 “NOT RECOMMENDED”，意味着可能在特定情况下有适当理由，该项条款可以接受或更适用，但必须充分理解其完整含义并仔细权衡之后，再做决定
MAY	“MAY” 或形容词 “OPTIONAL”，意味着该项条款确实是个可选项。产品团队可能因用户需求或觉得对产品有提升而选择该选项，同时其他产品团队可能忽略该选项



责任编辑 王照瑜 刘向杰

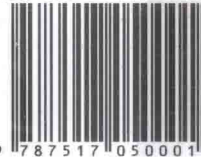
微信号: Waterpub-Pro



唯一官方微信服务平台

销售分类: 计算机基础

ISBN 978-7-5170-5000-1



9 787517 050001 >

定价: 90.00元

